Infrastructures and Compilation Strategies for the Performance of Computing Systems

Erven Rohou

HABILITATION À DIRIGER DES RECHERCHES

présentée devant

L'Université de Rennes 1
Spécialité : informatique

par

Erven Rohou

Infrastructures and Compilation Strategies for the Performance of Computing Systems

soutenue le 2 novembre 2015 devant le jury composé de :

Mme	Corinne Ancourt	Rapporteur
Prof.	Stefano Crespi Reghizzi	Rapporteur
Prof.	Frédéric Pétrot	Rapporteur
Prof.	Cédric Bastoul	Examinateur
Prof.	Olivier Sentieys	Président
M.	André Seznec	Examinateur

Contents

Chapter 1

Introduction

No other industry has boomed as fast as the computer industry. Hardware and software are everywhere and we rely on them every day. As a corollary of Moore's law[1], the performance of computers has evolved at an exponential pace, for decades! We now live in a world where billions of sensors, actuators, and computers play a crucial role in our life: flight control, nuclear plant management, defense systems, banking, of health care. This has solved many problems, created new opportunities, and introduced new threats.

Durig this long period of time, many things have changed: form factors, constraints, even users' expectations. What remains, though, is the need for performance.

In this document, we are interested in the performance of computer systems, and in particular how software is processed to run efficiently on hardware, and how it must adapt to continuously changing features. We discuss various aspects of compilation, interpretation, and optimizations.

This chapter briefly describes recent evolution of the hardware and software ecosystems, and presents our assumptions on future computing systems. The rest of the document describes our main contributions to the field of compilation.

1.1 Evolution of Hardware

Software runs on hardware. Ideally, hardware and software should evolve in synergy. Unfortunately, most of times, hardware evolves on its own. This is due to many constraints, involving technology, time-to-market, business opportunities, company organizations...) Eventually, software adapts.

[1] Gordon E. Moore is a co-founder of Intel in 1968. In 1965 he observed [Moo65] the exponential growth of the number of transistors on a chip. Performance is directly linked to the number of transistors.

1.1.1 Traditional Evolution

For decades, the performance of microprocessors has followed an incredible exponential pace (Moore's law). Since the 1970s, the performance has been doubling every 18 months. The Cray-1, first installed 1976, with its 80 Mflop/s is now outperformed by modern smart-phones, sometimes by an order of magnitude.[2]

The most advertised metric by silicon and PC vendors used to be the clock frequency. The clock contributes only partly to the performance increase, and improvements in micro-architecture also plays a significant role. Intel reported [BDK+05] in 2005 that the clock frequency represented 80 % of the performance gains to date. It is representative of the general trend of computing systems: from 20 MHz in 1990 to 2 GHz in 2000.

Other improvements in micro-architecture include, for example, pipelining, better cache hierarchy, branch prediction, prefetch, out-of-order execution, register renaming. They are transparent to the software in the sense that code can be functionally correct, even though it ignores the underlying details. Performance, however, can be impacted, and compilers must do their best to exploit the hardware at its best.

Evolution in architecture cannot be ignored, in particular additions to the instruction set. Each new generation of processor often comes with new features, such as floating point unit, SIMD extensions, or fused multiply-add (FMA). Code generation must adapt to take advantage of the potential additional performance, sometimes at a significant effort.

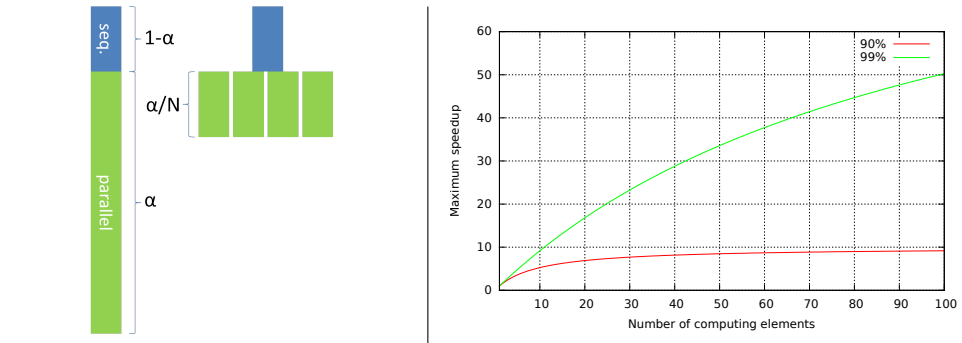
1.1.2 Multicore Era

Since 2002 however, despite considerable industrial effort, the frequency has been plateauing. Silicon vendors have hit a new *wall*: the "power wall". Power consumption has always been an issue for embedded systems. It became a concern for general purpose processors.

The dynamic power consumption is given by $P = \alpha V_{dd}^2 f$, where f is the operating frequency, and V_{dd} the voltage. Design constraints also make the voltage proportional to the frequency. As a rule of thumb, the power is a cubic function of the frequency. Intel's Fred Pollack illustrated the problem in a famous MICRO 1999 keynote [Pol99], comparing the power density (in W/cm^2) of processors to various objects, ranging from a hot plate to the sun's surface. The hot plate was approached by the PentiumPro and surpassed by the Pentium II manufacturing process at 0.5 μm, the nuclear reactor corresponds to 0.1 μm lithography. Current process targets 22 nm! Quite a number of techniques have been proposed to address the power wall, such as dynamic voltage and frequency scaling DVFS (aka Intel SpeedStep, Turbo Boost).

Moore's law, however, is still true. More and more transistors are integrated in processors. Advances in technology and micro-architecture now translate into more parallelism. The dual core was introduced in 2006, the quad core in 2008, the hexa

[2] Performance of supercomputers is typically evaluated by the Linpack benchmark. Android Apps are available to run it on smart-phones.



(a) Sequential and parallel sections (b) Maximum attainable speedup

Figure 1.1: Amdahl's Law

core was available in 2010. Prototypes have been demonstrated with up to 48 and 80 cores. Kalray announced the first samples of its MPPA 256 product in 2012. Intel's Xeon Phi processors feature 50+ physical cores.

High degrees of parallelism is not new. But they used to be for experts running scientific workloads on highly expensive supercomputers. Moderate amounts of parallelism now sell at the local supermarket.

1.1.3 Amdahl's Law

The consequences of this evolution on the software industry are dramatic: most existing software has been designed with a sequential model in mind, and even parallel applications contain residual sequential sections. The performance of these applications will ultimately be driven by the performance of the sequential part (as stated by Amdahl's law [Amd67]).

For simplicity, consider that a fraction α of a computation can be "perfectly" parallelized on N processors, and the rest is sequential (see Figure 1.1 (a)). The overall speedup is given by

$$S_N = \frac{1}{1-\alpha+\frac{\alpha}{N}} \quad \textit{and} \quad \lim_{N\to\infty} S_N = \frac{1}{1-\alpha}$$

Maximum attainable speedups for $\alpha = 0.9$ and $\alpha = 0.99$ are shown on Figure 1.1 (b). Assuming that 99% of an application is parallel – an overly optimistic hypothesis – speedup will never exceed 100×. While this is certainly a great speedup, 100 cores can only achieve half of it.

A more in-depth analysis of the implications of Amhdahl's law for symmetric and asymmetric multicores is developed by Hill and Marty [HM08].

1.1.4 Foreseeable Future Architectures

Several studies and technology roadmaps [ABC+06, Com07, DBLM+07] predict that it will be feasible to integrate thousands of cores on a chip by 2020. Amdahl's law makes symmetric many-core processors less attractive for many applications, because such a high number of cores will expose the sequential bottleneck. At the same time, Intel estimates [BDK+05] that 10-20 % of applications are easily parallelizable, 60 % with effort, and the rest very hard.

There are a number of factors that help us envision what future computing environments will be.

Asymmetry Due to Amdahl's law and the amount of legacy sequential software, we envision that future architectures will consist of many simple cores for efficient execution of parallel sections, but also a few very aggressive complex cores to execute sequential sections.

Dark Silicon Technology makes it relatively easy to manufacture a large number of transistors. But, due to power constraints, they cannot all be used at the same time. The large available silicon area makes it sensible to specialize parts of the chip for dedicated application domains (e.g. DSP, GPU, various accelerators) as long as they are not activated simultaneously.

Yield Manufacturing technology imposes another constraint. Design shrinks comes with increasing process variability. This means that it becomes increasingly difficult to guarantee that processors produced on the same manufacturing line are strictly identical. Large die areas make this effect even more pronounced. Vendors still need to maintain an acceptable *yield* (volume of "good" products), and thus sell most of the processors. A solution is to sort all products by functionality and performance (a process called *binning*) and sell them at difference price ranges.

All these factors call for a large diversity of upcoming processors, as well as the impossibility to predict what their characteristics will be, even few years from now.

1.1.5 Extremely Complex Micro-architectures

Micro-architectures have become extremely complex, and they usually do a very good job at executing fast a given sequence of instructions. When they occasionally fail, however, the penalty is severe. Pathological behaviors often have their roots in very low-level details of the micro-architecture, hardly available to the programmer.

For example, high-end processors have dedicated hardware for floating point computations. Some Intel processors however handle denormal numbers [IEE08] thanks to a sequence of micro-operations. It is described as "extremely slow compared to regular FP execution" in the Intel Optimization Reference Manual [Int14]. We observed that a simple computation on the x86 architecture can perform up to 87× worse than expected for pathological parameters, because of micro-operations [Roh12]. A real world application suffers a 17× slowdown.

Another example relates to the SSE and AVX SIMD extensions of x86 architectures. Due to very low-level design details, mixing them results in severe penalty, typically 75 cycles [Int14]. A compiler is unlikely to produce such a situation, however, it may happen with hand-written code or when linking with legacy libraries.

These are only two examples. Intel released [Int14] a 600+ page document, filled with techniques to help programmers get the most from their processors.

1.2 Evolution of Software Ecosystem

The amount of software-controlled systems has never been so large. Systems of medium complexity typically consist of millions of lines of code. Legacy is more important than ever.

Thanks to binary compatibility, the continuous "automagic" increase of performance has been a very good news for software developers and users. In 1998, Todd Proebsting estimated the respective contributions of hardware and compilation technology to the performance of computing systems. His findings, sometimes referred to as Proebsting's Law [Pro98, Sco01], show that compilers double the performance of the code they generate every 18 *years*. Nicholas FitzRoy-Dale repeated the experiment a decade later [FD09] and obtained slightly more pessimistic results.

Still, state-of-the-art static compiler typically provide 2× to 4× speedups when optimizations are enabled, and most programmers would not forego this additional performance, especially when it comes for free. Exceptions include developers of real-time systems where the worst-case execution time is more important than the average case. In this field, optimizations may degrade the estimate because they lose critical semantic information.

Others [Pug00] have proposed to take advantage of the extra performance to recover the penalty induced by higher-level programming models and languages.

Above all, what this seemingly pessimistic law hides, is that the compiler community has been able to keep up with all architectural and micro-architectural evolutions, as well as the increasing complexity of programming languages (e.g. C++). Research in compilation during these years produced many optimizations targeted at various aspects of the architectures and micro-architectures [Muc97, App98, ASU88]. For example, deeper memory hierarchies were introduced to deal with the so-called "memory wall" because the memory subsystem could not sustain fast increase of clock rates. Numerous articles have been published on codes transformations that deal well with caches and prefetch.

1.2.1 Evolving Ecosystem

Programming parallel systems used to be for computer scientists, or experts in specific domains (such as physicists developing demanding applications for supercomputers). The most visible evolution of hardware in the last decade is the introduction of parallelism in commodity systems. Everyone now gets a parallel system, at the local store, for a few hundred euros. Yet, parallel programming is considerably more difficult than sequential programming.

In parallel to this evolution, many scientific disciplines have developed a digital dimension [Inr12] (computational medicine, computational biology, computational archaeology...) Domain scientists develop applications in less traditional languages, favoring productivity-friendly environments (Python, R, Matlab...), but still run computationally heavy workloads.

A steady trend is that the lifetime of many software is long, much longer than the hardware for which they are initially designed. If binary portability (i.e. the portability of the functionality) is likely to be preserved, the portability of the performance is at risk. Contrarily to what we have been used to (performance increases with the clock frequency at each new generation of hardware), performance will no longer scale automatically with the number of cores.

Processors are not the only moving target of computing systems, the entire computing environment becomes dynamic. Two relatively recent paradigms drastically changed the ecosystem: the cloud, and mobile *Apps*.

Cloud computing consists in using computing resources provided by a vendor, typically at a remote location, over a network. Computation can be seen as a utility — similar to gas, water or electricity. The advantage for users is the ability to provision resources (and pay for them) only when needed. The drawback is that actual hardware on which an application runs is rarely known beforehand, and often shared with other users. In the (likely) case these nodes are not identical, developers target the least capable node, and often run under-optimized code. Even during execution, other jobs can start and compete for resources at any time.

Smart-phone *Apps* constitute another very diverse and rapidly moving target for application developers.

1.2.2 Extremely Complex Software Stacks

Taking advantage of the available compute power, software has also grown extremely complex. Real-world systems consist of many layers: hypervisor, operating system, runtimes, libraries, compilers... Each of these layers is a considerable effort. GCC, for example, has grown to more than 15 million lines of code, excluding libraries, LLVM has a few millions. Operating systems are even larger: Linux is approaching 20 million lines, while Microsoft Windows is likely closer to 100 million lines.

1.2.3 Summary

To summarize, the evolution of processors recently started changing in new ways. the most radical one being the introduction of parallelism to general purpose processors. Parallelism is now for everybody, not just HPC experts. The characteristics of the architecture we envision for the mid-term future are the following: highly parallel, with many simple cores (thousands of cores will be feasible by 2020); heterogeneous, with a few very aggressive sequential cores, and various accelerators. They will be very diverse, and it will also be difficult to anticipate the characteristics of the next generation from a current state-of-the-art. Applications running on top of these processors will be large, probably old, and largely unoptimized for the current target.

1.3 Executing Code

1.3.1 Static Compilation

Compilation can be broadly defined as the translation of a program from a high-level language to a low-level language. "High" and "low" are obviously loosely defined. The C language is high-level for embedded system and operating systems developers where some pieces of code are still written in assembly language. For developers used to object-oriented programming, software components, and functional languages, the C language will appear rather low-level, and used only to address system level layers. In most of this document, compilation refers to the translation of the C language to assembly, or bytecode to assembly (sometimes, C to bytecode).

In these terms, writing a compiler for a simple language targeting a simple assembly language is a moderate effort. This is a typical assignment for groups of final-year students at engineering schools. Compilers, however, must provide much more than a simple translation, including debug support, separate compilation, packaging (libraries), profiling... And finally, compilers also optimize the code to exploit the underlying hardware as well as possible. As mentioned before, industry-grade compiler are made of several million lines of code.

Compilation technology goes back to the 1950's. Since then, an incredible body of work has been produced, from low-level code generation techniques to supporting very high-level programming languages. Entire books have been written to describe the optimizations that transform a *correct* program into a *fast* program. Optimizations are the building blocks of a compiler. Much effort has produced a wide spectrum of techniques. However, since the 1990's, except for specialized processors with dedicated instruction sets, few optimizations were able to yield more than a few percents of performance. Moreover, combining several optimizations, or running them in the wrong order, often degrades performance. And even when they do well, there is often room for improvement.

1.3.2 Alternatives to Static Compilation

Static compilation is only one way to produce executable code. To facilitate the deployment of applications to diverse targets, the community has adopted bytecode representations. The ability to run Java applets in browsers made bytecode popular in the mid-1990s. Bytecodes are not natively executable by a processor, some mechanism is necessary. Several approaches exist. Historically, interpreters have been first proposed: they are simple to develop, easily portable to different hosts, but they are slower than native execution. Bytecodes can also be compiled to native code, on the target machine. This is sometimes referred to as *load-time* compilation: the entire application is translated at user invocation. The resulting code is close to what a static compiler would have produced, but execution incurs an initial overhead. Alternatively, *install-time* compilation processes the bytecode when it is first deployed, and the system keeps a native representation on non-volatile storage. Just-in-time compilation provides an interesting trade-off where the bytecode is compiled when needed, but at the granularity of a function. Overheads

are limited to processing a function, instead of the entire application, and this offers the opportunity to recompile functions over time at different optimization levels. Finally, regardless of how the native code has been produced, it can still be re-optimized by dynamic binary optimizers.

Interestingly, interpreters have seen a renewed interest in the recent years, due to the spread of languages such as Python, whose dynamic nature makes JIT compilation complicated.

1.4 Contributions

This document presents our main contributions to the field of compilation, and more generally to the quest of performance of computing systems.

It is structured by type of execution environment, from static compilation (execution of native code), to JIT compilation, and purely dynamic optimization. We also consider interpreters. In each chapter, we give a focus on the most relevant contributions.

Chapter 2 describes our work about static compilation. It covers a long time frame (from PhD work 1995–1998 to recent work on real-time systems and worst-case execution times at Inria in 2015) and various positions, both in academia and in the industry.

My research on JIT compilers started in the mid-2000s at STMicroelectronics, and is still ongoing. Chapter 3 covers the results we obtained on various aspects of JIT compilers: split-compilation, interaction with real-time systems, and obfuscation.

Chapter 4 reports on dynamic binary optimization, a research effort started more recently, in 2012. This considers the optimization of a native binary (without source code), while it runs. It incurs significant challenges but also opportunities.

Interpreters represent an alternative way to execute code. Instead of native code generation, an interpreter executes an infinite loop that continuously reads a instruction, decodes it and executes its semantics. Interpreters are much easier to develop than compilers, they are also much more portable, often requiring a simple recompilation. The price to pay is the reduced performance. Chapter 5 presents some of our work related to interpreters.

All this research often required significant software infrastructures for validation, from early prototypes to robust quasi products, and from open-source to proprietary. We detail them in Chapter 6.

The last chapter concludes and gives some perspectives.

Chapter 2

Static Compilation

In this chapter, we are interested in the translation of source code (written in a language such as C, C++) to assembly. The basic process involves parsing the input program, usually building an intermediate representation, and emitting assembly code. However, compilers doing just that produce poor code, similar to what most compilers produce when no optimization is applied (optimization level `-O0`). Optimizations are the key to performance.

Hundreds of optimizations have been developed over the last decades, and are covered in the literature [ASU88, Muc97, App98]. Each optimization tries to improve performance by focusing on a particular aspect: removing spurious dependencies, eliminating redundant computations, improving usage of resources...

Unfortunately, many problems are NP-complete – register allocation [CAC+81, BDGR07] and alias analysis [Hor97] in many cases, or instruction scheduling [HG83], to name a few. Carefully tuned heuristics play a critical role in the efficiency of compilers.

Moreover, the order in which optimizations are applied also drastically impact performance. This is well known in the community as the *phase ordering problem*. And to make things worse, many optimizations are parametric (unrolling factor, size of tiles, inlining depth...) Exhaustive exploration of all cases is simply intractable.

Our research was not about proposing new optimization techniques. Instead, our approach consisted in considering the large set of existing optimizations, and in proposing alternative ways to benefit from them. Section 2.1 presents *on-demand alias analysis*, our proposal developed in the context of an industrial compiler to limit the cost of memory disambiguation to where it is really needed by optimizations. We then introduce GCDS in Section 2.2: a study that takes *global* interactions in consideration, a domain where compilers typically under-perform. Section 2.3 introduces *iterative compilation*, a pioneering step to overcome the phase ordering problem when long compilation times are acceptable. Finally, we contributed to the field of real-time systems, by proposing to trace necessary source-level annotations down to binary code through optimizations (Section 2.4). Section 2.5 discusses perspectives and impact.

2.1 On-demand Alias Analysis

On-demand alias analysis was developed while at STMicroelectronics, in collaboration with Politecnico di Milano and Harvard University. Other participants include Marco Garatti, Roberto Costa, Stefano Crespi Reghizzi, Marco Cornero. Details can be found in the following publications:

- [GCCRR02] Marco Garatti, Roberto Costa, Stefano Crespi Reghizzi, and Erven Rohou. The impact of alias analysis on VLIW scheduling. In *ISHPC*, 2002. LNCS 2327.
- [CGRCR04] Roberto Costa, Marco Garatti, Erven Rohou, and Stefano Crespi Reghizzi. Hardware parameters of VLIW cores and code quality factors affecting alias analysis impact. *ST Journal of Research*, 1(2), 2004.

VLIW stands for *Very Long Instruction Word*. It refers to a class of processor architectures where instructions are scheduled entirely statically, and execution is fully in-order. This drastically reduces the complexity of the hardware, compared to out-of-order engines. The burden of extracting instruction level parallelism is thus pushed onto the compiler, making memory disambiguation of utmost importance.

When optimizing code, compilers often need to check whether two pointers refer to the same location. When they do, they are said to *alias*. Alias analysis (aka disambiguation) is the process of proving the relationship of two references. When we can show that they always refer to the same object, the relation is a *must-alias*. Conversely, if they are guaranteed to never refer to the same object, it is a *no-alias*. Otherwise, the relationship is classified as *may-alias*.

Alias analysis is an expensive analysis, both in terms of computation time and allocated memory. It needs to track all the possible values of pointers. For each pointer, it must identify all the possible objects that can be referenced. As the compiler optimizes the program, the internal representation changes, sometimes in radical ways. The analysis must then either continuously update its information, or recompute it regularly. Since alias analysis operates on pairs of pointers, the number of queries is also quadratic in the number of memory references.

We proposed *on-demand* alias analysis [GCCRR02] to limit the cost of computations, and illustrated it in relation with instruction scheduling for a VLIW processor. The alias analysis phase can be seen as a server, and optimizations needing information about memory references are the clients making the queries.

The performance of the generated code is dictated by the critical path of the data dependence graph (DDG) – the longest path between any two instructions. When a memory-induced dependence is on the critical path, the scheduler invokes the disambiguator. If such memory references can be proved *no-alias*, the corresponding edge is deleted, and the critical path shortened. Edges that are not on the critical path need not be checked, since deleting them would not contribute any shorter schedule. We proposed two heuristics to select when the disambiguator is invoked.

Static criterion: all queries are made on the original DDG, before the scheduler starts. With this criterion, we consider edges on the critical path, and invoke the analyzer as long as the length is reduced. The process then repeats with the new critical path. The queries are independent of the chosen scheduling

algorithm, because it operates on the original DDG. However, this approach cannot take resource conflicts into account and may produce less efficient schedules.

Dynamic criterion: queries are made while the scheduler progresses. When checking for ready-instructions, it also consider instructions that have only memory-related dependencies. Depending on the respective priorities of the instructions, we may invoke the disambiguator and try to delete an edge in the DDG.

Our analysis is intraprocedural and flow sensitive. It disambiguates structure fields, but not elements within an array (transformations involving arrays are typically performed at a higher level of representation, earlier in the compilation flow).

We first implemented [GCCRR02] our approach in the SUIF [WFW+94]/Machine SUIF [SH02] compiler infrastructure targeting a fictitious 4-issue VLIW processor (similar to the STMicroelectronics' ST200 family [FBF+00]). We experimented with several benchmark suites (SPEC 95, Mediabench, PtrDist). We observed that disambiguating only few edges, in the order of 20 %, benefit the scheduler. The dynamic criterion also generates much fewer requests, yet achieves nearly the same speedup. We also confirmed that only optimized code really benefits from alias analysis.

We also experimented [CGRCR04] our approach on top of the industrial, fully optimizing C compiler LxBE (described in Section 6.6.1), targeting the ST200. We considered two components of the compiler: partial redundancy elimination (PRE) [KCL+99] and three instruction schedulers: pre-scheduling is performed before register allocation to avoid false dependencies. Post-scheduling executes after register allocation to handles potential spill code. A modulo scheduler [LF02] also optimizes eligible loops. In addition to dependence information, the latter also requires *distance* information. Schedulers need *no-alias* information to delete spurious dependences; PRE needs *must-alias* to identify redundancy.

We studied the impact of architectural parameters, as well as the effect of compiler optimizations. Unsurprisingly, the general trend is that more hardware resources make alias analysis more beneficial. Additional load/store units make it more important, as well as increased issue width. Even though the ST200 already features a sizable register file, additional registers (we experimented with up to 512 registers) reduce the amount of spill code and thus decrease the impact of alias analysis, but only marginally.

Many compiler optimizations remove redundant computations and cause memory instructions to be on the critical path. In this case, alias analysis contributes to the final performance by letting the schedulers move them easily across each other. Certain optimizations eliminate memory instructions (such as scalarization, or load-store elimination). In such cases, the impact of alias analysis is clearly reduced. Overall, disabling optimizations in the compiler halves the speedup brought by alias analysis, from 5.2 % down to 2.7 % on the reference ST200, and from 9.1 % down to 4.4 % on a fictitious 8-wide machine with 512 registers and two load/store units.

2.2 GCDS – Global Constraints Driven Strategy

GCDS was developed at IRISA, within the framework of the FP5 European project OCEANS. Other participants include André Seznec, François Bodin and Christine Eisenbeis. Details can be found in the following publications:

- [RBES00] Erven Rohou, François Bodin, Christine Eisenbeis, and André Seznec. Handling global constraints in compiler strategy. *IJPP*, 28(4):325–345, 2000.
- [Roh98] Erven Rohou. *Infrastructures et Stratégies de Compilation pour Parallélisme à grain fin.* PhD thesis, University of Rennes 1, Nov 1998.

Compilers excel at local optimizations; but they have a hard time making global decisions. Traditional compilers process fragments of an application at various granularities (files, functions, regions, basic blocks...), but sequentially. Some compilers apply inter-procedural or link-time optimizations to re-optimize some parts of the program, but goal is to propagate information, such as constants, across classical optimization boundaries. And while ordering of optimizations has received a lot of attention, global interactions have remained mostly unexplored. Still, fragments are not – by far – unrelated. Many metrics, including code size, execution time, or register pressure, depend on the inter-relations of many fragments. It happens frequently that improving a metric on one fragment degrades another metric on the same fragment or on another one.

The structure of conventional compilers is not appropriate for dealing globally with issues such as code size and execution time. A built-in strategy locally applies a set of heuristics on code fragments in order to optimize the execution time. However, most optimizations that attempt to improve performance also increase code size [BGS94], therefore increase the overall instruction cache footprint of the program. Some code generation decision which seemed to *locally* conduct to a performance increase may result in a *global* net performance loss due to a raise of instruction cache misses. Final code generation decision should be taken globally rather than locally!

We proposed a global approach named GCDS (Global Constraints Driven Strategy) [RBES00], and we demonstrated it on the global trade-off performance vs. code size: relevant code fragments are identified through profiling and each fragment is optimized several times, in different ways. The individual characteristics of all resulting code are measured and fed to a global solver that optimizes for one criterion under constraints (see Figure 2.1). We focused on finding the best possible performance given a maximum code size, but other trade-offs can be possible.

We experimented with the Philips TriMedia TM-1000 [Cas94], a 5-issue VLIW processor dedicated to high-performance multimedia applications, featuring guarded instructions, latencies of one, two, three, and seventeen cycles, and a 3-cycle delay slot. Benchmarks consist in video decoders for two standards: *H263*, with six different input bitstreams; and MPEG2, with five different video sequences.

We selected five standard optimization sequences for VLIW processors.

S_0 applies the minimal set of transformations: local scheduling followed by register allocation.

U_n applies loop unrolling (n times), followed by superblock formation, and elimination of conditional jumps thanks to the insertion of predicates. Instruction scheduling and register allocation finalize the code generation.

U'_n is similar to U_n but register allocation is performed before scheduling. This decreases the code compaction potential, but usually requires fewer registers.

SP applies software pipeline to single basic block loops. [Lam88, RBES00].

I causes the function calls to be inlined. The large resulting body is scheduled before register allocation is performed. Inlining results in a larger block with more potential parallelism. It can also enable further transformations prevented by the function call, for instance software pipelining.

For each optimized fragment, we measured the resulting code size directly on the assembly file. Performance is estimated thanks to a linear cost model, simple yet sufficient for VLIW architectures. Trips counts are collected from profiling information. The selection of the best overall optimization sequence for each fragment is modeled as an integer linear programming problem, fed into a Simplex solver.

The optimizations are implemented within our Salto framework (see Section 6.1). We used `lp_solve` to solve to ILP problem. We identified six loops from *H263* and five loops from *mpeg2play* which account for more than 40 % of the total running time.

Figure 2.2 shows the performance achieved by GCDS under various code size constraints on our benchmarks. For selected performance points of *mpeg2play*, we also report the selected sequence for each loop. Despite the small number of loops and transformations involved, a large number of optimal design points are produced. This reflects the intrinsic complexity of the interactions at play between optimizations, and hints at the difficulty to achieve good performance with only a local view.

To better emphasize the importance of considering global impact, we also report the results of the following experiments.

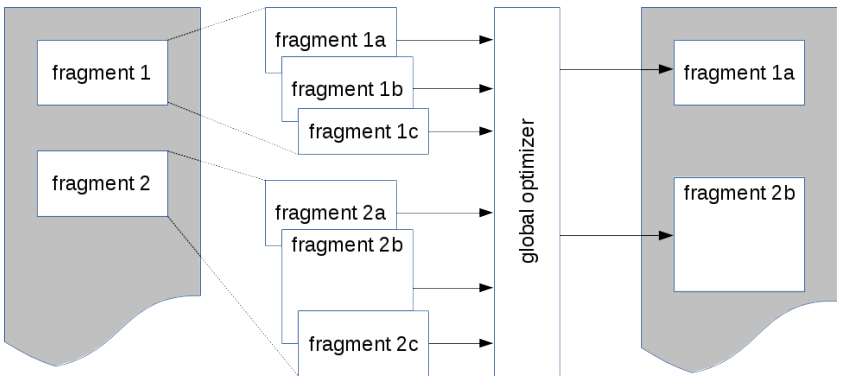
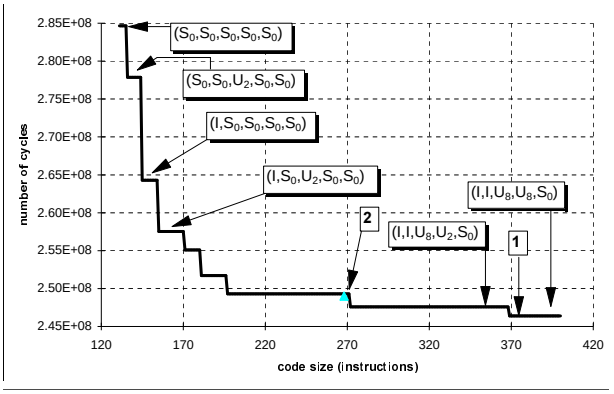
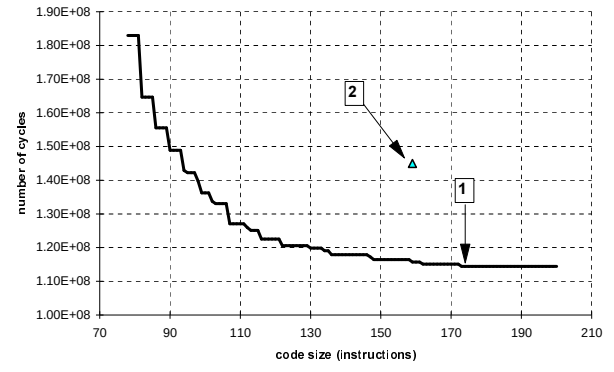


Figure 2.1: GCDS Principle



mpeg2play



H263

Figure 2.2: GCDS Results: best performance under code size constraint

- For each fragment, we select the best performing (local) optimization sequence, regardless of code size. Corresponding results are reported on Figure 2.2 with the label [1].
- For each fragment, we choose the transformation with the best asymptotic performance. No profiling is used, but loop bounds known at compile-time are exploited. Loop unrolling is constrained to choose higher unrolling factors only if they increase performance by at least 10 %. Lack of profiling information may cause inefficiencies, for example by invoking the modulo scheduler, despite the fact that the minimum number of iterations is not reached. This configuration is similar to a static compiler with local view only. It is reported as label [2] on Figure 2.2.

The analysis showed that even a very limited number of fragments is enough to generate very complex interactions. Using the GCDS approach, we were able to compute much more sensible solutions than the reference industrial compiler, for example by reducing the code size by 50 %, while maintaining the performance level at 99 % of the maximum.

2.3 Iterative Compilation

Iterative compilation was developed at IRISA, in collaboration with the University of Manchester and the University of Leiden, within the framework of the FP5 European ESPRIT project OCEANS. Other participants: André Seznec, François Bodin, Mike O'Boyle, Toru Kisuki, and Peter Knijnenburg. Details can be found in the following publications:

- [BKK+98] François Bodin, Toru Kisuki, Peter M. W. Knijnenburg, Mike F. P. O'Boyle, and Erven Rohou. Iterative compilation in a non-linear optimisation space. In *Workshop on Profile and Feedback-Directed Compilation, in conjunction with PACT*, 1998.
- [vdMRB+99] Paul van der Mark, Erven Rohou, François Bodin, Zbigniew Chamski, and Christine Eisenbeis. Using iterative compilation for managing software pipeline-unrolling tradeoffs. In *SCOPES*, 1999.
- [Roh98] Erven Rohou. *Infrastructures et Stratégies de Compilation pour Parallélisme à grain fin.* PhD thesis, University of Rennes 1, Nov 1998.

Compiler optimizations have been studied for decades. Ultimately, their goal is improve the performance of the code, based primarily on static analysis, possibly with the help of profiling information. However, actual performance depends on many fine grain details of the micro-architecture, and often also on input parameters. In the presence of hundreds of optimizations (some of them parametric), analytical models can hardly predict what the final performance will be, or even how to reach it.

We showed [BKK+98, Roh98] that two optimizations only are enough to produce a very irregular transformation space. Figure 2.3 represents how the execution time of matrix multiplication on the Philips TriMedia TM-1000 processor (z-axis) varies when loop unrolling and tiling change their parameters (respectively unrolling factor

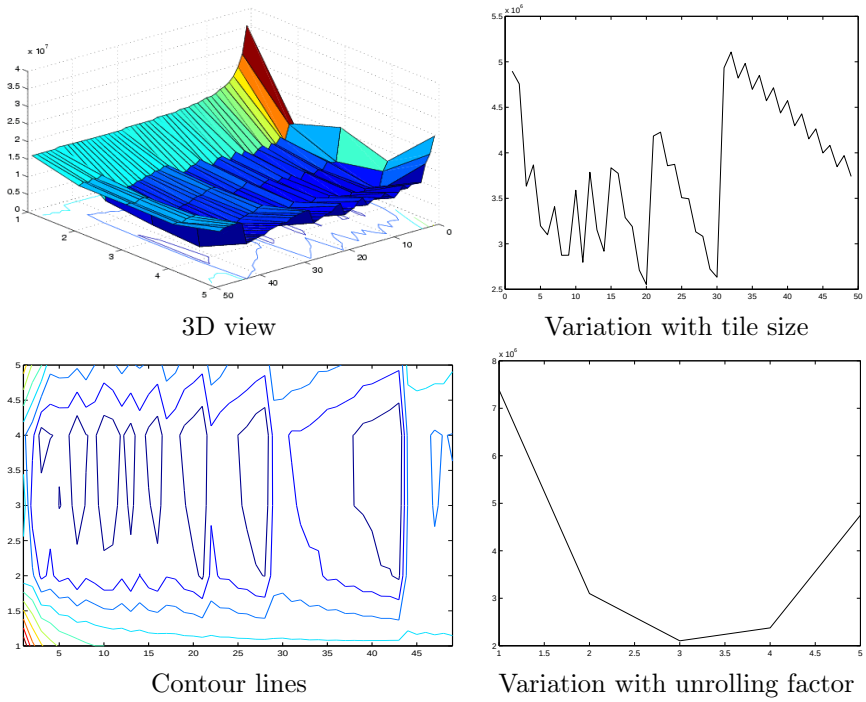


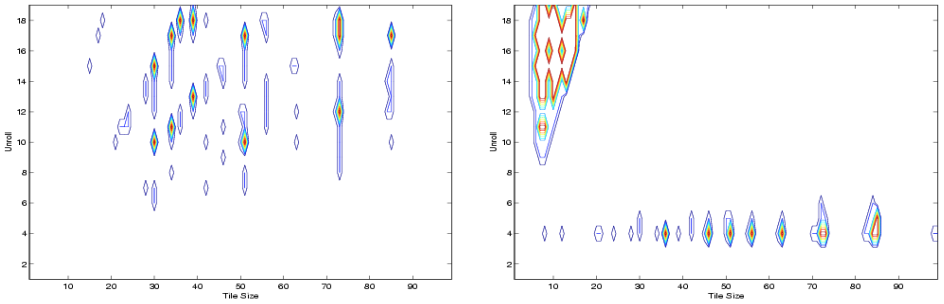
Figure 2.3: Iteration space on TM-1000, varying unrolling factor and tile size

and tile size). Such spaces are also highly dependent on the target architecture. As an example, Figure 2.4 graphically identifies the portion of the search space within 20 % of the absolute minimum, for two architectures: Intel PentiumPro and MIPS R10000. The PentiumPro has a more dispersed range of minima, the R10000 has most of its points close to the minimum.

Given the shape and the irregularity of the search space, static and analytical model can hardly find the best combination of parameters. We proposed *iterative compilation* to search through this space for "good" local minima by compiling and running several versions of an application, and using actual running time as a feedback. This is possible only if the compilation time constraint is relaxed, but this is typically not a problem in many scenarios, especially for embedded systems, scientific libraries, etc.

The search algorithm was not focus of our research. We used a simple algorithm that visits a number of points at spaced intervals. Points between the current global minimum and the average are added to a queue. Elements are iteratively taken from the queue, and their neighborhood is visited, at spaced intervals. The algorithm stops when a given number of points have been visited.

Figure 2.5 illustrates the convergence of our algorithm on the R10000 architecture. The behavior is identical on all architectures. In the case of the R10000, the absolute minimum is found in 150 iterations, but it reaches 10 % of the best per-



Pentium2
minimum at: unroll 17, tile 51

R10000
minimum at: unroll 4, tile 85

Figure 2.4: Local Minima for Intel PentiumPro and MIPS R10000

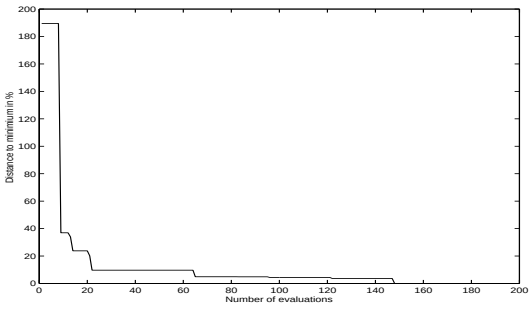


Figure 2.5: Successive performance points, for R10000

formance in 21 iterations. In the case of large transformation spaces, we achieved 0.3 % of best performance by visiting less than 0.25 % of the whole space, and we found the absolute minimum after visiting less than 1 % of space.

Existing static analyses are still very relevant, as they can provide initial seed points to start the iterative search.

We also showed that adding a third optimization – padding, with size from 1 to 10 – invalidates the findings of two optimizations. As an example, without padding, the best transformations on a Sun UltraSparc are for unroll = 3. When padding is also considered, however, none of the points within 20 % of the minima have an unroll factor of 3. This demonstrates the close connection of transformations and the error introduced when considering them separately.

We also applied iterative compilation [vdMRB+99] to the exploration of trade-offs between loop unrolling and software pipelining , and the combination of high-level transformations with low-level optimizations. This pioneering step in the late 1990s eventually developed into a rich field (see Section 2.5).

2.4 Real-Time Considerations

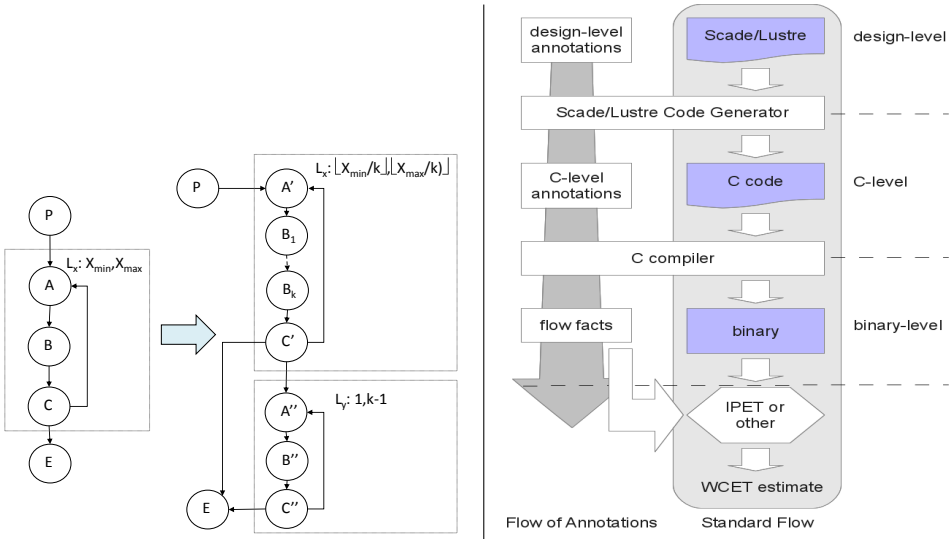
This research is being conducted at Inria, with the framework of the ANR project 12-INSE-0001 WSEPT, and the PhD of Hanbing Li. Other participants include Isabelle Puaut. Details can be found in the following publications:

- [LPR14] Hanbing Li, Isabelle Puaut, and Erven Rohou. Traceability of flow information: Reconciling compiler optimizations and WCET estimation. In *RTNS*, 2014.
- [LPR15] Hanbing Li, Isabelle Puaut, and Erven Rohou. Tracing flow information for tighter WCET estimation: Application to vectorization. In *International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, 2015.

Real-time systems have become ubiquitous. For this class of systems, correctness implies not only producing the correct result, but also doing so within specified timing constraints. Designers are required to estimate the worst-case execution time (WCET) of their systems to guarantee that all applications meet their time constraints. Many WCET estimation methods have been proposed, operating through static code analysis, measurements, or a combination of both. By definition, the estimate must be *safe*, i.e. it must be an upper bound of the time required to execute a given task on a given hardware platform, in any condition. To be useful, WCET estimates shall be as *tight* as possible, i.e. as close as possible to the actual worst-case. This second property is useful to avoid over-provisioning the system.

WCET estimation tools take into account any possible flow of execution in the control flow graph, regardless of actually semantically feasible flows. Additional information that limits the set of possible flows of control (the so-called *flow information*) improves the tightness of WCET estimates. Flow information, either produced automatically or inserted manually through annotations, is typically inserted at source code level. On the other hand, WCET analysis must be performed at machine code level. Most tools propagate high-level information to low-level code by building the corresponding control flow graphs and matching them. Between these two levels, hundreds of compiler optimizations – some very aggressive – may have a dramatic effect on the structure of the code, resulting in the impossibility to match the control flow graphs, hence a loss of useful information. See, for example, the impact of loop unrolling alone on Figure 2.6 (a). For this reason, many existing WCET tools for real-time systems turn off compiler optimizations when computing WCET.

We proposed [LPR14] a framework to trace and maintain flow information from source code to machine code to benefit from optimizations, yet improving the WCET estimates. Instead of considering the compiler as a black-box and trying to match its input and output, we modified each optimization of the compiler to systematically update flow information. What is crucial is that transforming the flow information is done *within* the compiler, in parallel with transforming the code (as illustrated in Figure 2.6 (b)). There is no *guessing* what flow information have become, it is transformed along with the code they describe. Back to Figure 2.6 (a), X_{max} being the maximum trip count of the original loop, we can guarantee that the first



(a) Impact of loop unrolling on the CFG (b) Information flows in the compiler

Figure 2.6: Traceability of flow information

loop (unrolled k times) iterates maximum $\lfloor X_{max}/k \rfloor$ and the second loop (handling remaining iterations when X_{max} is not a multiple of k) iterates maximum $k-1$ times. In case the optimization is too complex to update the information, we always have the option to drop it. The result would then be safe, even though it would probably result in a loss of precision. However, we can also notify the programmer (or the compiler developer) that this optimization causes problems in a real-time context, making it possible to disable it. Note that only problematic optimizations must be disabled, as opposed to all of them in most current real-time systems.

Our framework considers flow information as linear constraints on the execution frequency of basic blocks. For each code optimization, it defines a set of rules that specify how constraints must be updated to reflect the transformed code. The implementation currently handles loop bounds, and we support most optimizations of LLVM, including aggressive transformations such as vectorization [LPR15].

Our implementation in the LLVM compiler shows that we can improve the WCET estimates of Mälardalen benchmarks [GBEL10] by 60 % on average, and up to 86 %. We also provide new insight on the impact of existing optimizations on the WCET.

Future work might consider even more complex optimizations, such as the polyhedral model. Another interesting directions consists in investigating how to trace other kinds of flow information, such as infeasible (mutually exclusive) paths.

2.5 Impact and Perspectives

By definition, static compilation cannot take into account varying input, or varying hardware. At most, a compiler can rely on multiversioning to handle critical versions: it then generates several versions of the most important pieces of code. The approach is however inherently limited by the code size explosion. Auto-tuning helps chose the best parameters for performance among predetermined values. Profiling can be used to specialize for a given (set of) input(s). This is effective when the chosen inputs have similar characteristics, but not reliable when the relevant inputs vary too much. From an industrial point of view, profiling is also complicated to integrate in production cycles: it requires running the workloads. The teams in charge of developing applications and doing the final integration of all software components are often different. Profiling feedback would incur a feedback also at the level of the company, between teams, with a significant organizational overhead, and thus cost.

Our research on static compilation did not attempt to develop new optimizations, or try to address particular features of architectures. Instead, our approach consisted in considering optimizations as building blocks of compilers, taking a higher level view, and tackling complex interactions of optimizations.

We showed that the cost of memory disambiguation can be lowered by focusing the analysis to the dependences that matter. In spite of the industrial environment, we obtained and published a few new results about alias analysis [GCCRR02, CGRCR04] for VLIW processors.

Iterative compilation was pioneering work that eventually yielded to a rich field, still explored today (see for example the outcome of European projects such as ACOTES [MAB+11]). To the best of our knowledge, we wrote the first publication on this topic [BKK+98] (111 citations according to Google Scholar, at the time of writing). Since then, this approach has been extended and applied to various contexts. The cluster Adaptive Compilation of the European Network of Excellence HiPEAC is partially dedicated to this topic. The Interactive Compilation Interface (ICI) is also a descendant of the initial iterative compilation proposal. It lets compiler developers interact with the compiler internals and experiment with new strategies. ICI was used in the EU project MILEPOST. It is supported by many partners, both academic and industrial (Chinese Academy of Science, NXP, Mandriva, Univ. of Edinburgh, INRIA) The Nano2012 and Nano2017 Mediacom collaborative projects between STMicroelectronics and INRIA also includes iterative compilation among the promising approaches to be considered. It is also a component of an ANR proposal currently under submission.

Interestingly, iterative compilation was motivated by the difficulty to accurately predict performance on a particular processor. The situation is much worse today than in 1998, making the approach still very relevant. In fact, the topic is still being explored, better search algorithms have been proposed, as well as coupling with machine learning techniques. Guillon et al. [GMVK13] at STMicroelectronics are also developing an approach to reduce the cost of the exploration, yet outperforming the production compiler on performance and code size.

Hardware, on the other hand, has the visibility on runtime conditions. Aggressive out-of-order processors schedule instructions based on accurate information. The downside is that they have limited visibility on the future: the reorder buffer only contains at best a few hundred instructions. Still, out-of-order processor tend to make compiler optimizations less critical than on VLIW processors. Unfortunately, they also make the behavior unpredictable and performance assessment difficult. An extreme case is reported by Hundt et al. [HRTV11] where the mere insertion of `NOP` instructions improves performance by visible amounts.

Just-in-time compilation and dynamic binary optimization provide interesting points to (re-)optimize applications. We discuss our contributions in Chapters 3 and 4 respectively.

Chapter 3

Just-in-Time Compilation

Just-in-time (JIT) compilation refers to a process where the generation of machine code happens just before it is needed – that is, after the execution of the program has started, from an input program not in native binary form. This is the typical execution setup of Javascript in web browsers (sent in source format), or Java (sent in Java bytecode).

JIT compilation goes back to the 1960's. A brief history can be found in Aycock [Ayc03]. According to him, the first occurrence is due to McCarthy for his work with Lisp on IBM 704. Significant successful works include Smalltalk in the 1980's, and Self. But JIT compilation really blossomed with the release of Java [CFM+97]. The ability to deploy programs expressed in bytecodes as *applets* that run in a browser, regardless of the underlying processor and operating system was the key enabler for this technology. Since then, a number of bytecodes and JIT technologies have appeared, for example: CLI (supported by Microsoft's .NET as well as Mono, ILDJIT, Portable.NET), LLVM, Google's NaCl, or more recently Facebook's PHP JIT.

When a program is deployed in bytecode, a JIT compiler is in charge of translating bytecode to machine code as needed. The resulting code is placed in a so-called *code cache*. Figure 3.1 illustrates a high-level view of such a system. At its first invocation, a function f is compiled. Functions called by f, however, are not compiled, and call sites are replaced by *trampolines*, i.e. calls to the runtime system. Whenever the call happens, the system will take over, generate machine code, fix the call site, and resume execution. A function that is never called is never compiled. The name *code cache* derives from the fact that this location behaves as a hardware cache: due to the limited size, older entries must be evicted to make room for newer ones.

JIT compilation has a number of advantages over classical (or AOT – Ahead of Time) compilation.

- Only executed functions are compiled, thus reducing the memory footprint, and the overhead of code generation.

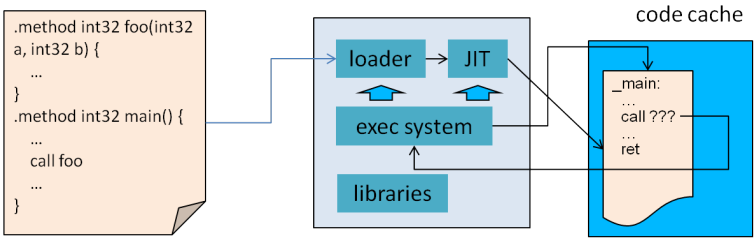


Figure 3.1: Code cache

- Profiling information can be collected before actual compilation happens, for example by relying on an interpreter for the first few calls (see for example Sun/Oracle's HotSpot [PVC01]). Collected data then drives the compilation heuristics.
- Actual parameter values are observable. Functions can be specialized, without the overhead of multiversioning, because pseudo-constants are known.
- Actual hardware is known, making it possible to generate the best possible code for a particular architecture, instead of binary compatible code that can be deployed at large, but is under-optimized on most machines.
- Re-optimization is possible. This is interesting when the context of optimization changes (such as pseudo-constant parameter that drove function specialization changes). Conversely when a function has been called a large number of times, it also makes sense to re-optimize at a higher optimization level: there is hope to recoup a higher overhead when the better function will be called many times in the future.

On the flip side, JIT compilation also has drawbacks.

1. Compile-time is now part of the user-visible run-time, and memory is also shared between the JIT compiler and the application being run. This is somewhat mitigated by the fact that only hot functions are optimized, and the empirical "90/10" rule (90 % of the time is spent in 10 % of the code) promises that the overhead will be well spent. Still, JIT compilers are limited in the aggressiveness of the optimizations they apply.
2. The system as a whole is more dynamic and less predictable: machine code is produced at the user/customer end, hence difficult to validate. The limited size of the code cache may force evictions, and future re-compilations. These events depend on execution patterns at the user/customer end, and are hardly predictable. This is a very serious limitation for real-time systems.
3. Programs to be executed under the control of a JIT compiler are typically deployed in bytecode format. The most widespread bytecodes (Java, CLI, LLVM) rely on strongly typed, high level representations of the program that

is fairly readable, easily reverse engineered, and reveal much more information than plain machine code. This may be a threat for industrials trying to protect their intellectual property, and a number of obfuscation techniques have been developed to mitigate this risk.

In the rest of this chapter, we describe three contributions related to the drawbacks of JIT compilation. First, split-compilation (Section 3.1) is a high-level proposal targeted at unleashing the potential of JIT compilation by taking the best of the two worlds: offline and online stages. We illustrate it on split-vectorization, a concrete application to vectorization: a notoriously complex yet powerful optimization. Second, we present a first step towards reconciling JIT compilation and real-time systems in Section 3.2. Third, we show how dynamic code generation and frequent recompilation can turn JIT compilers into an advantage to harden code obfuscation (Section 3.3).

3.1 Split-Compilation

This research has been started at STMicroelectronics, then pursued at Inria, in particular within the context of the Nano2012 and Nano2017 collaborative programs. Other participants include: Albert Cohen, David Yuste Romero, Kevin Williams from Inria, and Dorit Nuzman, Ayal Zaks, Sergei Dyshel, Ira Rosen from IBM. Details can be found in the following publications:

- [CR10] Albert Cohen and Erven Rohou. Processor virtualization and split compilation for heterogeneous multicore embedded systems. In *DAC*, 2010.
- [NDR+11] Dorit Nuzman, Sergei Dyshel, Erven Rohou, Ira Rosen, Kevin Williams, David Yuste, Albert Cohen, and Ayal Zaks. Vapor SIMD: Auto-vectorize once, run everywhere. In *CGO*, 2011.
- [RDN+11] Erven Rohou, Sergei Dyshel, Dorit Nuzman, Ira Rosen, Kevin Williams, Albert Cohen, and Ayal Zaks. Speculatively vectorized bytecode. In *HiPEAC*, 2011.

We proposed split-compilation [CR10] as a way to mitigate some of the shortcomings of JIT compilation. Leveraging the multi-stage compilation (such as source-to-bytecode followed by bytecode-to-native, but other variants are possible), the key idea is to split the compilation process in two steps — offline and online, and to off-load as much of the complexity as possible to the offline step to keep the online step as lean and efficient as possible. Figure 3.2 illustrates this principle.

Offline This step occurs on the developer's workstation. Resources are virtually unlimited. Ideally, all the work should be done here. However, target and execution environment are unknown. Target-dependent optimizations are impossible – or risky at the best. Dynamic events are also unknown (program inputs, competing processes on a server), causing missed opportunities.

Online This step occurs on the user's device, at run-time. In case of an embedded system, resources are much more limited that the offline step. But the actual target system (hardware and computing environment) is known, enabling much more focused and precise optimizations.

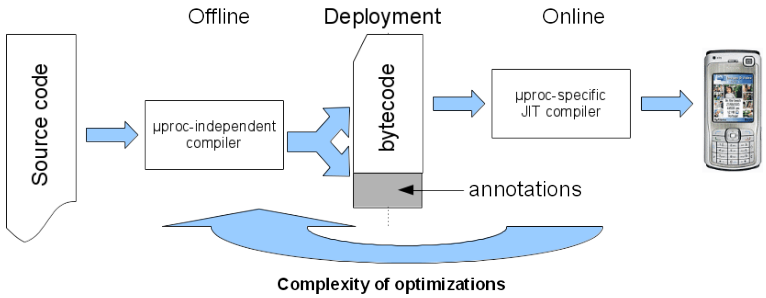


Figure 3.2: Principle of split-compilation

```
char a[N], b[N], c[N];
for(i=0; i<1024; i++) {
  a[i] = b[i] & c[i];
}
```

(a) Scalar code

```
char a[N], b[N], c[N];
for(i=0; i<1024; i+=4) {
  a[i..i+3] = b[i..i+3] & c[i..i+3];
}
```

(b) Vectorized (pseudo) code

Figure 3.3: Vectorization

Vectorization at a Glance. In brief, vectorization is a compiler optimization. It consists in recognizing patterns of repetitive operations applied to adjacent values and packing them into larger containers (i.e. *vectors*). The optimized code applies the same operation in parallel to all elements of the vector. This is also referred to as SIMD parallelism (Single Instruction Multiple Data). As an example, consider the code of Listing 3.3. The original loop applies the bitwise `AND` (&) operator to all elements of arrays b and c. This is illustrated on the left part of Figure 3.4. The optimized loop (pseudo C code in Figure 3.3-b) loads four elements of each array at a time in a regular 32-bit register, applies the bitwise operation on the 2×4 elements, and stores the four elements. Note also the new loop increment (i+=4).

Advantages of vectorization include fewer executed instructions, fewer memory accesses that take advantage of the wide memory bus, and fewer loop iterations. All silicon vendors now provide dedicated instructions – SIMD extensions – to manipulate vectors.

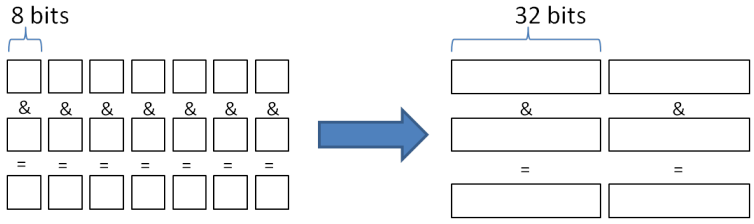


Figure 3.4: Vectorization illustrated

Vectorization can deliver high speedups when the conditions are met. Unfortunately, SIMD extensions are very diverse, in terms of vector width, supported data types, alignment constraints, supported idioms. This makes it difficult to design a multi-platform auto-vectorizer. Vectorization also relies complex and expensive data dependence analyses, as well as preparation code transformations. As an illustration, GCC-4.9 has 32,000 lines of code only for the vectorizer proper. This motivates a split compilation approach in which the identification of vectorization opportunities and their safety conditions are prepared offline.

Split-vectorization. We proposed Vapor SIMD [RDN+11, NDR+11], developed in collaboration with IBM Haifa Research Lab. The bytecode is speculatively vectorized, in a way that is straightforward for a JIT to generate native SIMD instructions when available, yet easy to revert to scalar code when SIMD is not available.

Being a split-compilation approach, the crux of Vapor SIMD is to move as much complexity as possible from the online stages to the offline ones. Offline stages are responsible for all target independent optimizations; expensive analyses can be run, and the results encoded in the bytecode. Online stages later use these encodings to both reduce compilation time and increase code quality. Aggressive offline stages address the difficulties of automatic vectorization. Online stages allow for fine adjustments to the actual instruction set.

Our abstraction layer is built on top of the CLI format. It consists in idioms that can be translated into efficient code on any targeted SIMD ISA, yet encompass the whole spectrum of instruction sets, targeting the greatest common denominator of all platforms. We handle alignment constraints, interleaving, initialization of constant vectors, scatter/gather operations, widening, and reduction operations of basic arithmetic and more complex ones such as dot products [NDR+11]. The offline stage is implemented in GCC4CLI, our port of GCC for the CLI format (see Section 6.6.2). As far as online stages are concerned, we considered two kinds of execution systems: JIT compilers, including Microsoft .NET and Mono's open source solution, as well as a static compiler. The purpose of the latter is to compensate for Mono's limited capabilities, a more recent and less mature software than static compilers. For this, we relied on a CLI front-end for GCC that converts CLI to GCC's internal representation and the leverages all the GCC optimizations.

We define a *conscious-JIT* as a JIT that is aware of the naming convention used by the offline compiler to convey vectorization opportunities, and can generate efficient SIMD instructions accordingly. Conversely, an *agnostic-JIT* is any JIT that can process the same standard bytecode but is unaware of this naming convention or cannot make use of it to generate SIMD instructions (e.g. due to lack of hardware support).

Our flow is illustrated on Figure 3.5. We experimented with various architectures and JIT technologies. Configuration A deals with scalar code and constitutes our reference. Conscious JITs (configuration B) consist in:

- Mono for Linux on Intel Core2 Duo;
- Mono for Linux on PowerMac, with SIMD support enabled.

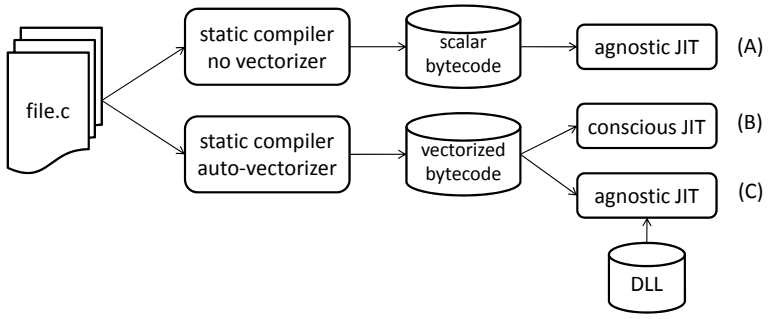


Figure 3.5: Scenarios: (A) regular flow; (B & C) Vapor SIMD flows

Agnostic JITs (configuration C) include:

- Microsoft .NET on Windows XP Pro on Intel Core2 Duo;
- Mono on Linux with SIMD support (SSE) disabled on Intel Core2 Duo;
- Mono on Linux for TI UltraSPARC;
- Mono on Linux for PowerMac with SIMD support (AltiVec) disabled.

Since agnostic JITs are unaware of the newly introduced naming convention, we provide an implementation in a separate dynamic library. For this purposed we relied on `Mono.Simd.dll` which we extended to cover all needed idioms.

Risk. We first validated that our approach is risk-free, i.e. agnostic JITs incur only minor penalties. Figure 3.7-(a) reports the performance (higher is better) of configuration C with respect to configuration A. These numbers confirm that the penalty of running vectorized bytecode through an agnostic-JIT is limited. Moreover, in many cases performance even improves.

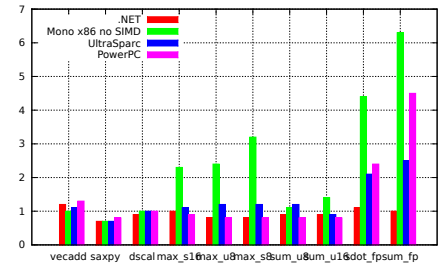
Looking at overall averages, only the .NET platform exhibits performance degradation of 7 % on average. The internals of the .NET platform are not documented and its performance is difficult to analyze. However, the JIT is reported to have a low code size threshold for inlining [Nor03]. Even though the arithmetic operations in the DLL are small, they might not be inlined.

Several kernels compute reductions. The effect of scalarizing vectorized reduction code (with the arithmetic operation inlined) is similar to loop unrolling followed by modulo variable expansion (MVE). The code sample of Figure 3.6 illustrates this effect. Here we can see the critical path in the scalar loop is the circular data dependence on the accumulator. In the scalarized loop, it has been split into four independent components, thus improving performance. However, this transformation may result in aggressive unrolling (the char and short kernels are unrolled 16 and 8 times respectively) spilling of intermediate values, and in turn additional memory traffic. On PowerPC this additional memory traffic is the key factor behind

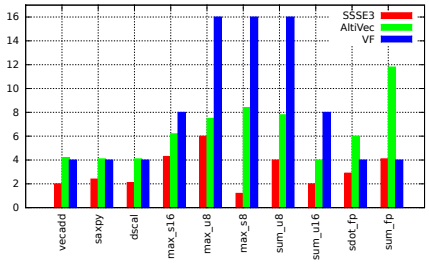
```
for (i=0; i<n; ++i) {
  s += a[i];
}
```

```
for (i=0; i<n; i+=4) {
  s0 += a[i];
  s1 += a[i+1];
  s2 += a[i+2];
  s3 += a[i+3];
}
s = s0 + s1 + s2 + s3;
```

Figure 3.6: Effect of unrolling and modulo variable expansion



(a) Performance of (scalarized) vectorized bytecode on agnostic-JIT



(b) Speedup with conscious JIT

Figure 3.7: Performance of agnostic and conscious JIT compilers

the performance degradation of the integer kernels. The x86 platform is less sensitive to the additional memory traffic, but benefits from the relaxed dependencies due to the MVE effect.

Efficiency. The efficiency objective refers to performance improvements delivered by a conscious-JIT when SIMD support is available. Figure 3.7-(b) reports how the conscious JITs perform, along with the vectorization factor.

The observed speedups on PowerPC are mostly in line with the expected speedups from vectorization: they are comparable to the Vectorization Factor (VF), minus the usual overheads of vectorization (handling misaligned accesses, initializing vectors of constants, reduction prologue and epilogue, etc). However, code generation issues related to the actual JIT used in the experiment offset the results in some cases.

The vectorization impact on PowerPC is between VF/2 and VF with the exception of a few super-linear speedups. The vectorized reduction kernels suffer from the lack of global vector register allocation in Mono, which results in loading and storing of the reduction variable in each iteration of the vectorized loop. This explains the reduced speedups on the integer reduction kernels. Mono also does not perform global register allocation for (scalar) floating point registers, resulting in even more redundant loads and stores in the scalar code than in the vectorized code, which explains the super-linear speedups

Similar trends are observed on the x86 platform, however scaled down by half. Again, this is due to Mono's poor register allocation capabilities: more variables

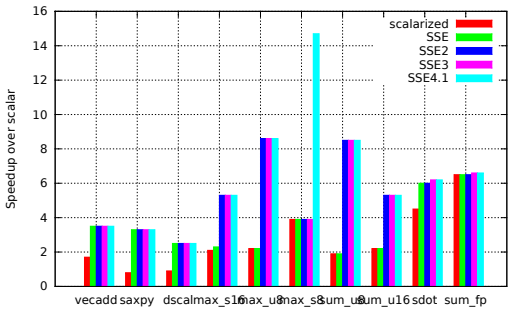


Figure 3.8: Speedup with conscious JIT, depending on SSE level

are needed in the vectorized version (e.g. additional induction variable to advance addresses by the vector width), which on x86 with its few available registers results in register spilling. These code generation issues are not inherent to JITs in general, and work is undergoing on Mono's development branch to improve its register allocator. This would allow the benefit of vectorization to manifest itself also on targets that do not have many registers, such as PowerPC.

Future-proof. One of the motivations of our work is to support evolutions of architectures. We take a closer look at the behavior of the same vectorized bytecode on successive versions of the SSE family: SSE, SSE2, SSE3, SSSE3, SSE4.1 and SSE4.2. Figure 3.8 summarizes the run times of all kernels for each supported SSE level. Relative performance is computed as the speedup over the scalar implementation.

The main observation is that speedups increase monotonically with the SSE level. The JIT compiler automatically takes advantage of the available hardware support to provide acceleration, and falls back to scalarized code for unsupported features. This is especially visible in the case of *max_s8* and *max_u8*, which are identical except for the signedness of the max operator. The signed max on packed bytes is introduced only in SSE4.1, whereas unsigned max first appeared in SSE2.

The *horizontal add* instruction was added in SSE3. It is used by *sdot* and *sum_fp*, but only for the final reduction in the loop epilogue after the main computation loop, hence the limited impact on performance. The code size, however, is reduced.

Summary. We have proposed split-compilation as a way to facilitate the adoption of aggressive optimizations in the context of constrained JIT compilers. By splitting the optimization flow into two separate steps – offline and online – we can are able to make powerful analyses and transformations viable at runtime. We demonstrated our approach in the notoriously complex case of vectorization. Not only can vectorized bytecode be efficiently processed by both agnostic and conscious JIT compilers, but the performance also scales with the evolution of SSE extensions.

3.2 Real-Time Considerations

This research was developed at Inria, in the context of the internship of Adnan Bouakaz. Other participants include Isabelle Puaut. Details can be found in the following publications, as well as in Adnan's final report [Bou10]:

- [BPR11] Adnan Bouakaz, Isabelle Puaut, and Erven Rohou. Predictable binary code cache: A first step towards reconciling predictability and just-in-time compilation. In *RTAS*, 2011.
- [EEMR+14] Sara Elshobaky, Ahmed El-Mahdy, Erven Rohou, Layla AA El-Sayed, and Mohamed Nazih ElDerini. A lightweight incremental analysis and profiling framework for embedded devices. In *SCOPES*, 2014.

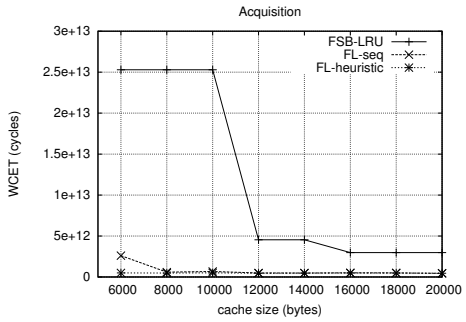
Virtualization and just-in-time (JIT) compilation have become important paradigms in computer science to address application portability without deteriorating average-case performance. Unfortunately, JIT compilation raises predictability issues, which currently hinder its dissemination in real-time applications. We proposed [BPR11] to start reconciling the two domains, i.e. to take advantage of the portability and performance provided by JIT compilation, while providing some predictability guarantees.

As a first step towards reconciling JIT compilation and real-time systems, we focused on the behavior of the code cache (illustrated in Figure 3.1, in previous section). Its size is limited. Similarly to a hardware cache, when no room is available for a new entry, some other entries must first be evicted. Note that arbitrarily complex cache structures and replacement policies can be used since the cache is implemented in software. In our case, cache entries are entire functions. Because they are of different sizes, memory fragmentation might be introduced.

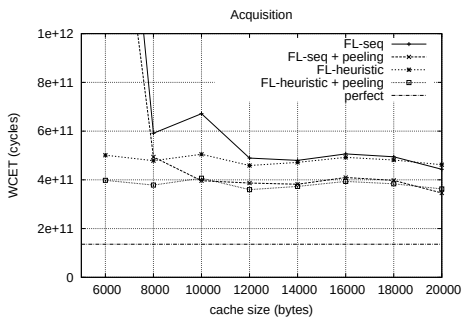
We study two structures of code caches and demonstrate their predictability. On the one hand, the studied binary code caches avoid too frequent function re-compilations, providing good average-case performance. On the other hand, and more importantly for the system determinism, we show that the behavior of the code cache is predictable: a safe upper bound of the number of function re-compilations can be computed, enabling the verification of timing constraints. We explored two cache structures.

Fixed-size blocks with LRU replacement (FSB-LRU). For this first structure, the code cache is decomposed in fixed-size blocks. The block size is equal to the size of the largest binary code of all program functions. The rationale behind the selection of a fixed size for cache blocks is to eliminate external fragmentation, possibly at the cost of increased internal fragmentation. The replacement policy for this cache structure is the LRU (Least Recently Used), selecting the least recently executed function in case of eviction. LRU is known to be the most amenable to accurate analysis [RGBW07].

Fixed layout cache (FL). In this second structure, every function is assigned a start address ahead of time. It is computed such that functions with heavy caller-callee relationships do not overlap in the code cache. Upon a cache miss



(a) WCET for each cache structure



(b) WCET for fixed-layout cache and perfect cache (close-up)

Figure 3.9: WCET estimate (cycles) for varying cache size (Task *Acquisition*)

when calling, or returning to, a given function f, all functions that conflict with f are evicted from the cache, before f is compiled and its binary code inserted in the cache. This structure requires the address range of every functions to be computed. We developed two methods for that purpose. In the first one (*FL-seq*), functions are statically assigned an address range through a sequential scan of the functions according to their declaration order. It ignores caller-callee relationships and is only used as a baseline. In the second method (*FL-heuristic*), we order functions according to their potential for WCET reduction, and we scan them in decreasing order. Each function is placed, trying to minimize conflicts with already placed functions.

We experimented our technique on the Debie software [HLS00] that monitors space debris and micro-meteoroids *in situ* by detecting impacts using mechanical and electrical sensors. The performance metric is the WCET estimate of every task. We estimate it using the state-of-the-art technique IPET [PS97] (Implicit Path Enumeration Technique).

Execution times for basic blocks are computed as follows. First, for the sake of simplicity, we assume that each bytecode instruction executes in one cycle. In particular, hardware cache effects are not taken into account (hardware caches have an orthogonal impact and can be handled by extensions of classical techniques [TFW00]). Second, when a call or return instruction is found in a basic block, the analyses classify it as a hit or a miss. In the latter case, a compilation time is added to the time value of the basic block containing the call/return instruction. We opted for a compilation time of the form $a \times x + b$, where x is the size of the function in bytes, and a and b are two constants characterizing the JIT compiler. The rationale behind this formula is that compile time can be bound by a start-up time and compilation speed (a and b respectively). JIT compilers run under severe constraints and cannot afford any non linear algorithm.

Figure 3.9 illustrates the performance of each cache structure on a representative task (*Acquisition*). Full results are presented in our RTAS publication [BPR11].

Figure 3.9 (a) illustrates all the proposed structures: fixed-size blocks with LRU (*FSB-LRU*), fixed-layout sequential (*FL-seq*) and heuristic (*FL-heuristic*). Figure 3.9 (b) does not show *FSB-LRU* but focuses on a reduced range of cache sizes to magnify lower-order phenomena. It also depicts the WCET estimate assuming a *perfect* code cache (the compilation time of every function is accounted for exactly once).

General observations. As expected, the WCET estimate globally decreases when the cache size increases. Nevertheless, there are some irregularities in the behavior of the layout determination heuristics. They are explained by threshold effects: to lower the algorithmic complexity, the heuristics approximate conflict costs between functions. At the same time, functions cannot be split in the code cache. A single byte difference in the cache size may relocate a number of functions and have drastic effects. This is especially true at small cache sizes.

We observe that for all benchmarks and most cache sizes, the *FSB-LRU* structure performs worse (i.e. yields higher WCET) than the fixed-layout structure. This comes from two factors:

- *Memory fragmentation*: the block size of *FSB-LRU* is the size of the largest function, thus introducing potentially large internal fragmentation in blocks when function sizes are very heterogeneous;

- *Replacement strategy*: in the *FSB-LRU* structure, the cache replacement policy (LRU) is fixed and independent of the application call patterns. In contrast, *FL-heuristic* structures compute the layout ahead of time based on some knowledge of the application. They have an opportunity to adapt the cache replacement to the application call pattern.

Comparing heuristics. In Figure 3.9 (b), we observe that *FL-heuristic* provides improvements over the baseline *FL-seq* for small cache sizes. For large sizes, the two layout generation methods behave identically, because the cache is large enough to store all functions after they are first compiled.

However, for three out of six tasks, even a large cache does not result in a WCET estimate comparable with a perfect cache. This is due to the presence of conditional constructs within loops. Even if the loop is peeled (see below), none of the functions called in the conditional construct can be guaranteed to be in the cache after the first iteration, and thus both functions are classified as miss. This phenomenon, well known when analyzing hardware caches, has a deeper impact on our analysis because of the granularity of cache entries.

Loop peeling. When a function is called from a loop, it often happens that the first iteration will result in a miss, and all following iterations in a hit. Because of the initial miss, the call must be globally classified as a miss. Loop peeling improves the analysis by moving the first iteration, including the missing call, outside the loop, increasing the likelihood that all references from the loop are hits. Loops

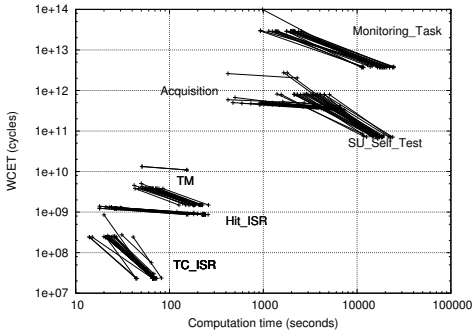


Figure 3.10: WCET estimate vs. computation time (layout generation + fixed-point analysis)

are *virtually* peeled, for analysis purposes only, no physical loop peeling is undertaken. Figure 3.9 (b) shows the impact of peeling all outermost loops on the WCET estimates.

All tasks benefit from peeling, but its effectiveness clearly depends on the tasks code structure (number of function calls within outer loops). For instance, the improvement for *Acquisition* is lower than for the other tasks, because there are fewer function calls in loops. Improved tightness is obtained at the cost of higher computation time. Figure 3.10 illustrates this trade-off: all WCET data points of Figure 3.9 (b) are represented with their computation time (in seconds). The left-most part of segments represent analyses without peeling (faster to compute, looser WCET estimate), the right-most part represents the same analysis with peeling (slower, but tighter estimate). The six clusters denote the six tasks. We observe that peeling always improves the WCET estimate, sometimes by more than an order of magnitude. Computation time, however, can also increase significantly.

Summary. JIT compilation is certainly not appropriate for critical real-time systems. However, for soft real-time systems, where occasional deadline misses only impact the quality of service, we have made a first step towards reconciling the need for predictability with the apparent non-determinism of JIT compilers. We showed that the behavior of the code cache can in fact be modeled using techniques similar to what is used for hardware caches. The total number of cache evictions and recompilations can be bound, thus improving the tightness of WCET estimates.

3.3 Just-in-Time Obfuscation

This research was developed at Inria, in collaboration with the Egypt-Japan University of Science and Technology, thanks to a collaboration grant PHC IMHOTEP. Other participants: Ahmed El-Mahdy, Muhammad Hataba, Marwa Yusuf. Details can be found in the following publications:

- [YEMR13] Marwa Yusuf, Ahmed El-Mahdy, and Erven Rohou. On-Stack Replacement to Improve JIT-based Obfuscation – A Preliminary Study. In *International Japan-Egypt Conference on Electronics, Communications, and Computers*, 2013.
- [HEMR15] Muhammad Hataba, Ahmed El-Mahdy, and Erven Rohou. OJIT: A novel obfuscation approach using standard just-in-time compiler transformations. In *International Workshop on Dynamic Compilation Everywhere*, 2015.

Distributed and ubiquitous software is becoming an integral part of the everyday life. On the business side, more and more companies rely on grid computing and cloud services. Sales related to Software as a Service (SaaS), as well as Infrastructure as a Service (IaaS) and Platform as a Service (PaaS) are expected to grow significantly in the near future. On an individual side, smart-phones are widespread; small, focused, user-developed applications (often referred to as *Apps*) are blooming. Software has always been under attack, its omnipresence will make software security of utmost importance.

The HiPEAC Vision [DYDS+10] identifies Computer Games "as one of the future killer applications for high-end multi-core processors [...] and one of the driving forces for our industry". Gaming is facing considerable piracy issues, but all categories of software are strongly impacted. The considerable development of the software industry, and the growth of new markets (cloud computing and smart phone Apps) all over the world will only make the situation worse.

In December 2010, the Gartner Group [Hal10] estimated that SaaS sales would reach $9bn (up 15.7 % from 2009), and predicted stronger growth in 2011, up to $10.7bn (a 16.2 % increase). In parallel, the Business Software Alliance [Bus11] estimates that 42 % of software was pirated worldwide in 2010, a $59 billion commercial value.

Attackers study the internals of programs, statically, but also often dynamically, relying on debuggers, disassemblers and tracing tools to get a sense of the working of a program. Possibly, they also generate a repeatable attack, to be shared with other attackers.

Obfuscation [CN09] has been proposed as possible countermeasures, the motto being "security by obscurity". Software vendors deliberately produce intricate software, introducing useless variables, opaque predicates, or contorted control flow to make it more difficult to comprehend and prevent tampering or reverse engineering.

Obfuscating JIT. We proposed OJIT (*Obfuscating JIT*) [HEMR15] to leverage JIT compilation to make software tamper-proof. The goal is to take advantage of the JIT technology and the development of multicore systems to improve the security of software. The central idea is to make reverse engineering more difficult by

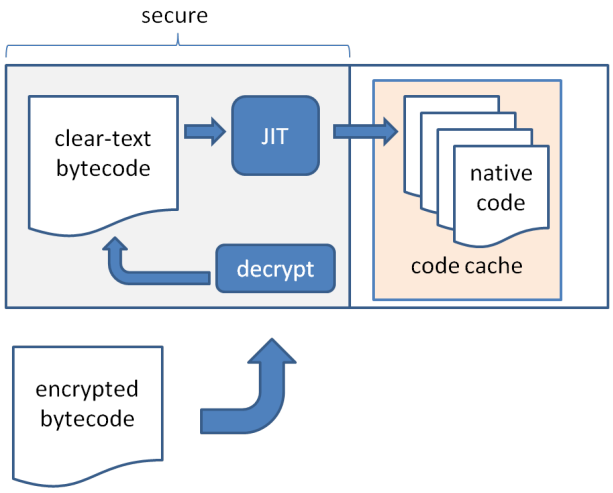


Figure 3.11: Possible use scenario for OJIT

presenting constantly evolving binary code to the user. We leverage JIT compilation to continuously modify the code under execution, and multicores to hide latencies.

Traditional JIT compilers rely on the code cache to store executable machine code. Functions are kept as long as possible in the cache, and evicted only when needed: either because a new, optimized, version has been produced, or when the capacity is exceeded and room is needed for a new function. Conversely, we propose to never keep functions in the cache and recompile a function at each invocation. The choice and order of optimizations, as well as parameters when relevant, is driven by a random number generator. A strong random number generator [SS03] guarantees that generated code is not reproducible though the functionality is the same. This presents a moving target to an attacker, making reverse engineering more difficult. The diversity of running binary code also renders ready-to-use attacks inefficient.

Improved OJIT with OSR. Using this approach, switching points only occur at function boundaries. In case of long running functions, this may hinder security. We further extended our technique to make switching possible at any basic block boundary, at the binary level. We built upon the *On-Stack-Replacement* techniques already proposed [FQ03] by mature JIT compilers, and we also proposed [YEMR13] a new technique that does not need inserting special code in the compiled program (i.e. no safe points).

OJIT generates continuously changing machine code. An attacker could circumvent the protection by inspecting the bytecode. A possible way to address the possibility is illustrated on Figure 3.11: the bytecode produced by a software vendor is encrypted before being deployed. Decryption happens on a secure server, protected, e.g. by a TPM module. Clear-text bytecode remains in the secure part of the server, only obfuscated native code can be observed by an attacker.

Performance is not sacrificed thanks to multi-core architectures: the JIT runs on separate cores, overlapping with the execution of the application. When overall performance is too severely impacted, obfuscation can also be limited to critical functions, such as checking a license key or a password. Alternatively, overhead can also be limited by recompiling functions less often, or only at selected checkpoints, thereby introducing a trade-off between security and performance.

Summary. Many JIT compilers take as input a bytecode, typically a high-level strongly typed representation of the program to be run. This representation is often fairly readable, which makes reverse engineering easier. Despite this fact, we showed that JIT compilers can also be used to reinforce security by increasing software diversity. The key idea is to use standard optimizations as black boxes to produce many variants of the same function, and the JIT compiler as the engine.

3.4 Impact and Perspectives

Split-compilation is a powerful concept that opens many interesting directions. It has already been applied to register allocation [DCRC10]. Many classical optimizations can be revisited in this context, pushing as much of the work as possible to the offline step. Several other scenarios are also possible, beyond mere splitting of optimizations.

- In many cases, the analyses that guarantee the validity of the transformation are expensive. For example, proving that a loop is parallel is complex and expensive. On the one hand, parallelism may not be exploitable on the actual target device. On the other hand, the result is one bit of information, which can easily be conveyed from the offline to the online steps. The loop may also be parallel only under some runtime condition, which can be reduced to a simple test.
- The offline compiler can help focus the JIT's work. It may, for example, point to places where the code generation was inefficient because it was lacking information, and suggest ways to improve performance depending on specific runtime conditions.
- Building on top of iterative compilation (discussed in Section 2.3), an offline compiler can explore how optimizations impact performance depending on particular hardware features, such as number of registers or available instruction level parallelism. Results of the lengthy but offline exploration are then encoded as *recipes* in the deployed bytecode, to focus the JIT compiler on a promising optimization strategy.

These directions will be explored within the framework of the Nano2017 PSAIC research program, involving Inria and STMicroelectronics. Split-compilation is a lively research field: in his keynote address [1] at CGO/HPCA 2014, Norm Rubin

[1] `http://cgo.org/cgo2014/wp-content/uploads/2013/05/Keynote-Norm.pdf`

mentions it as a "new way to look at optimization". His next example even considers vectorization. The techniques presented are based on a synergy between the static and the JIT compilers. In the next chapter, we will present techniques based on purely dynamic binary rewriting to adapt vectorized code to the underlying hardware.

JIT compilers introduce flexibility in the code generation and optimization process, at the cost of additional non-determinism. Real-time systems *need* worst-cases execution time guarantees. Computing tight bounds on reasonably complex processors is already a daunting task. Adding a JIT compiler in the system adds a new dimension to the complexity. Still, we made a first step in the direction of reconciling these two worlds. We extended existing techniques dedicated to the analysis of hardware caches, and we showed that we can analyze the behavior of the code cache, and bound the number of recompilations.

Instead of trying to statically estimate the execution time of the JIT, we can leverage the slack time due to the pessimistic bounds to execute the JIT. As soon as the allocated time slot expires, the JIT must instantaneously stop working, and resume at a later time, when a new opportunity window opens. We started exploring this incremental analysis and optimization within the PhD of Sara ElShobaky at Alexandria University: an initial proposal has been presented at SCOPES 2014 [EEMR+14].

Finally, we addressed one of the shortcomings of many JIT compilers: the fact that programs can be easily reverse-engineered due to the readability of many bytecode formats. Instead, we showed that the JIT engine combined with the diversity of optimizations can also be a powerful obfuscation tool.

Chapter 4

Dynamic Analysis and Optimizations

Dynamic analysis and optimization refers to the capability of observing a running program, collecting information, and re-optimizing parts of the application while it runs.

4.1 Motivation

Dynamic optimization is similar to the JIT compilation techniques presented in Chapter 3 in the sense that they operate at runtime. They are however very different because they operate on binary code. Much of the high level information present in source code or bytecode has have been lost by compilation passes, and many assumptions are implicitly made by the compiler, and not "documented" in the binary. Modifying programs in binary form at runtime seems like a radical idea.

This chapter first motivates dynamic binary optimization as a challenging but interesting and powerful operating point for optimizers. We then present two contributions: dynamic re-vectorization of binary code, and software memoization of pure functions.

4.1.1 Why Dynamic Optimizations?

In fact, runtime is a very natural operating point for an optimizer: the program is visible as a whole (including specific instances of libraries), source code is not needed (and neither are the compiler toolchains), making it possible to optimize commercial or legacy code, and finally the actual hardware is known. In addition, the analyzer observes a particular run, dictated by the very input data under execution, which may impact performance. It is well known that profile-guided optimizations must use carefully selected execution profiles to be effective. Input data impacts the regions of code that are executed (for example where several algorithms are implemented, as in MPEG2 vs. MPEG4 decoding), the size of data structures

(for example the size of images and videos to be processes) and consequently the pressure on the memory hierarchy, as well as the actual flow of computations (compressing a uniform vs. structurally complex image). We have even observed, that, in extreme cases, particular inputs can cause a 10× slowdown of floating point computations when denormal numbers are processed [Roh12].

Being able to analyze and optimize unmodified production code, even without rerun, is important in industrial setups. In particular, this approach does not require linking with any special library, using special compiler flags, or adding instrumentation.

4.1.2 Why Binary Optimizations?

At runtime, with the exception of interpreted workloads discussed in Chapter 5, only the executable in binary format is available[1]. Self modifying code (SMC) has been a known practice forever, and legacy code rely on it. This even forces vendors of the x86 architecture to maintain complex hardware that synchronizes the instruction and data cache. Dynamic binary optimization – and JIT compilers – can be seen as a more sophisticated form of SMC.

An alternative to binary optimization that does not involve SMC is function interception. This is a feature of the operating system that lets a user substitute library functions by their own functions, implemented in a different location.

We recently started exploring both approaches. Our tool Tiptop [Roh12] (described in Section 6.2) provides easy access to the behavior of an application wrt. the underlying micro-architecture. Our prototype Padrone [RRC+14] (described in Section 6.4) adds the capability to interact and modify executables.

The rest of this chapter first describes how we applied Padrone to the case of re-optimization of vectorized code, augmenting performance by replacing older SSE instructions by their more powerful AVX counterparts. We then illustrate an application of function interception for memoization, a technique where the return values of pure functions are cached to avoid future invocations.

4.2 Re-optimization of Vectorized Code

This research has been developed at Inria since 2012, within the context of the Inria Project Lab MULTICORE, and the PhD of Nabil Hallou. Other participants include Emmanuel Riou, Alain Ketterlin, Philippe Clauss. Details can be found in the following publications:

- [HRCK15] Nabil Hallou, Erven Rohou, Philippe Clauss, and Alain Ketterlin. Dynamic re-vectorization of binary code. In *SAMOS*, 2015.
- [RRC+14] Emmanuel Riou, Erven Rohou, Philippe Clauss, Nabil Hallou, and Alain Ketterlin. PADRONE: a Platform for Online Profiling, Analysis, and Optimization. In *Workshop of Dynamic Compilation Everywhere*, 2014.

[1] Dedicated techniques produce *fat* binaries that embed additional information, even the intermediate representation of the compiler. See Nuzman et al. [NED+13] for an example.

```
1  .L2:  movaps  A(rax),xmm0
2        addps   B(rax),xmm0
3        movaps  xmm0,C(rax)
4        addq    $16,rax
5        cmpq    $4096,rax
6        jne     .L2
```

(a) Original SSE

```
.L2:  vmovaps     A(rax),xmm0
      vinsertf128 1,A(rax,16),ymm0
      vaddps      B(rax),ymm0
      vmovaps     ymm0,C(rax)
      addq        $32,rax
      cmpq        $4096,rax
      jne         .L2
```

(b) Resulting AVX

Figure 4.1: Body of vectorized loop for vector addition

Applications are often under-optimized for the hardware on which they run. Several reasons contribute to this unsatisfying situation, including the use of legacy code, commercial code distributed in binary form, or deployment on compute farms: when targeting a range of different machines, developers must chose the minimum hardware configuration they require. This implicitly implies that more recent features will not be exploited. In fact, backward compatibility of instruction sets guarantees only the functionality, not the best exploitation of the hardware. In particular, SIMD instruction sets are always evolving.

We developed [HRCK15] a runtime re-vectorization platform that dynamically adapts applications to execution hardware. Programs distributed in binary forms are re-vectorized at runtime for the underlying execution hardware. Focusing on the x86 SIMD extensions, we are able to automatically convert loops vectorized for SSE into the more recent and powerful AVX. In the spirit of split-compilation, a lightweight dynamic mechanism leverages the sophisticated technology put in a static vectorizer and adjusts, at minimal cost, the width of vectorized loops. Since the binary is already vectorized, we are concerned only with the *conversion* of vector SIMD instructions from SSE into AVX, and some *bookkeeping* to guarantee the legality of the transformation. This is however different from Vapor SIMD (see Section 3.1) where static and dynamic components collaborate. In this approach the static compiler is unaware of the dynamic optimizer.

Figure 4.1 illustrates the general idea. It shows the loop body of a vector addition `C[i] = A[i]+B[i]` compiled to SSE assembly, and how we convert it to AVX. Register `rax` serves as a primary induction variable. In SSE, the first instruction (line 1) reads four elements of array A into the SSE register `xmm0`. The second instruction (line 2) adds in parallel four elements of B, and the third instruction (line 3) store the results into four elements of C. The induction variable is then incremented by 16 bytes: the width of SSE vectors (line 4).

Converting the loop body is relatively straightforward. However, guaranteeing the legality of transformation requires adjusting legality tests inserted by the static compiler, or adding new tests in some cases. Additional bookkeeping is also necessary to maintain the semantics of the optimized code.

Convert SSE instructions into AVX equivalent: in most cases, there is a one-to-one mapping between SSE and AVX instructions. In some cases, a single SSE instruction requires a few (typically 2) AVX instructions to implement the same semantics on a wider vector. Conversely, a single AVX instruction

is sometimes enough to capture the semantics of several SSE instructions, for example in the case of the fused-multiply-add (FMA). The `xmm` registers are converted to `ymm`.

Restore the state of `ymm` registers: in the (unlikely) case `ymm` registers are initially alive across the loop, the values of the registers we modify must be saved and restored.

Adjust strides of induction variables: AVX operates on 256-bit registers, while SSE handles 128-bit vectors. Induction variables used to access SIMD vectors must have their stride doubled.

Adjust trip count: The trip count of loops must be halved. Two cases are possible:

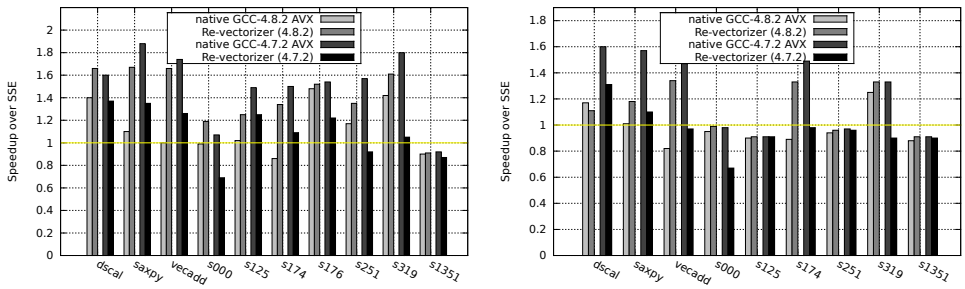
1. The trip count is statically known. In case it is even, it can be simply divided by 2. Otherwise, care must be taken to execute the remaining iteration properly.
2. The trip count is not statically known. In that case, it is already dynamically derived from the scalar trip count and the vectorization factor VF. The compiler has generated code to compute $n_{SSE} = \lfloor n/VF_{SSE} \rfloor$. We locate it and substitute VF_{SSE} by VF_{AVX}. Due to the fact that n is not necessarily of multiple of VF, remaining iterations are already handled.

Enforce data dependencies: The risk when increasing the vectorization factor is to exceed the iteration dependence distance. For example the simple loop with body `A[i] = A[i+4]` where `A` is an array of integers can be vectorized for SSE (VF=4), but not AVX (VF=8). We found this case to be rare in practice, but it is possible. Again, two cases are possible:

1. In the general case, the compiler emits a test to be executed at runtime to decide whether the vectorized code is legal. We locate this test and adjust it to check legality of AVX vectorization.
2. When array addresses are statically known, the static compiler could prove that the transformation is legal, and no test is generated. Because we operate at run-time, we can dynamically probe array addresses as well and check whether our conversion to AVX is also legal.

Handle alignment constraints: generally speaking, the x86 instruction set is flexible wrt. alignment constraints. Still, 16-byte aligned memory accesses may not be 32-byte aligned. When unsure, and the instruction requires vector-alignment, we generate two 16-byte aligned AVX instructions.

Our technique is implemented in the Padrone framework (see Section 6.4). We experimented with a 64-bit Linux Fedora 19 workstation featuring an Intel i7-4770 *Haswell* processor clocked at 3.4 GHz. Turbo Boost and SpeedStep are disabled in order to avoid performance measure artifacts associated with these features. We observed that different versions of GCC produce different behaviors. We experimented



(a) float (b) double

Figure 4.2: Speedups for type `float` and `double`. Reference is original SSE code.

with two relatively recent version of GCC: GCC-4.7.2 (Sep 2012) and GCC-4.8.2 (Oct 2013) available on our workstations, and two sets of benchmarks. The first consists in a few hand-crafted loops that illustrate basic vectorizable idioms. The second is a subset of the TSVC suite [MGG+11]. All TSVC kernels manipulate arrays of type `float`. We also manually converted them to `double` to enlarge the spectrum of possible targets and assess the impact depending on data types. We compile our benchmarks with GCC using the flags `-O3 -msse4.2` (`-O3` enables the GCC vectorizer).

We assessed the performance of our technique by means of two comparisons. First we measure raw speedup, i.e. we compare the transformed AVX-based loop to the original SSE-based. Then, we also compare it with the native AVX code generated by GCC with flags `gcc -mavx -O3`. Figure 4.2 reports the speedups of both native GCC for AVX and our re-vectorizer compared to SSE code. In the case of our re-vectorizer, we also report how it compares to native GCC AVX. These numbers are shown for data type `float` and `double`. We report results for both compiler versions. Taking `dscal` as an example, we show that the AVX code produced by GCC runs 1.4× faster than the SSE version. The code produced by our re-vectorizer runs 1.66× faster than the SSE version, that is a 19 % improvement over the AVX version.

We also visually confirmed that the difference in code quality between the SSE references produced by both compilers is small compared to the variations we observe between SSE and AVX.

GCC-4.8.2. As a general trend, our re-vectorizer is able to improve the performance of eligible loops, up to 67 %. More surprisingly, we also constantly outperform GCC for AVX, up to 66 % in the case of `vecadd`. There are two reasons to this:

1. When targeting AVX, GCC-4.8.2 generates a prologue to the parallel loop to guarantee alignment of one of the accessed arrays. Unfortunately, the loop does not take advantage of the alignment and relies on unaligned memory accesses (`vmovups` followed by `vinsertf128` when a single `vmovaps` sufficed). When targeting SSE, there is no prologue, and the loop relies on 16-byte

aligned memory accesses. In fact, AVX code generated by GCC-4.7.2 is more straightforward, without prologue, and similar to our own code generation. Corresponding performance also correlates.

2. GCC-4.8.2 tries to align only one of the arrays. Testing for alignment conditions of all arrays and generating specialized code for each case would result in excessive code bloat. The static compiler hence relies on unaligned memory accesses for all other arrays. Because we operate at runtime, we have the capability to check the actual values of the arrays addresses and generate faster aligned accesses when possible.

In the case of `s115`, we correctly identified the vectorized hot loop, but we were not able to locate its sequential counterpart in the function body, needed to execute a few remaining iterations when the trip count is not a multiple of the new vectorization factor. The reason is that the native compiler chose to fully unroll this epilogue. Our optimizer simply aborted the transformation.

The only negative effect occurs in s1351, with a 9 % slowdown. Note that the native AVX compiler also yields to a 10 % slowdown. The loop is extremely memory intensive, and increasing the vectors from 16 to 32 bytes reaches the maximum memory bandwidth of the machine.

Performance of our re-vectorizer as well as native GCC AVX is generally lower when applied to kernels operating on type `double`. The reason is that arrays with the same number of elements are twice larger, hence increasing the bandwidth-to-computation ratio, sometimes hitting the physical limits of our machine, as well as increasing cache pressure. We confirmed that halving the size of arrays produces results in line with the `float` benchmarks.

Two benchmarks failed with type `double`: `s115` and `s176`. This is due to a current limitation in our analyzer: the instruction `movsd` may be translated in two different ways, depending on the presence of another instruction `movhpd` operating on the same registers. Our analyzer currently considers instructions one at a time, and must abort. Future work will extend the analysis to cover such cases.

GCC-4.7.2. With GCC-4.7.2, our re-vectorizer sometimes degrades the overall performance compared to SSE code. We observe that this is particularly true when the same register (`ymm0`) is used repeatedly in the loop body to manipulate different arrays. This increases significantly the number of partial writes to this register, a pattern known to cause performance penalties [Int14]. This is particularly true in the case of `s125`. Despite these results, since our optimizer operates at runtime, we always have the option to revert to the original code, limiting the penalty to a short (and tunable) amount of time.

Compared to native AVX, as opposed to GCC-4.8.2, we systematically perform worse. This is expected because the native AVX compiler often has the capability to force alignment of arrays to 32 bytes. We only have the guarantee of 16-byte alignment, and we must generate unaligned memory accesses. The net result is the same number of memory instructions as SSE code, while we save only on arithmetic instructions.

Overhead. Overheads includes profiling the application to identify hot spots, reading the target process' memory and disassembling its code section, building a control flow graph and constructing natural loops, converting eligible loops from SSE to AVX, and injecting the optimized code into the code cache.

Profiling has been previously reported [RRC+14] to have a negligible impact on the target application. In addition, with the exception of code injection, all steps are performed in parallel with the execution of the application.

On a multicore processor, running the re-vectorizer on the same core as the target improves communication between the two processes (profiling, reading original code, and storing to the code cache) as the expense of sharing hardware resources when the two processes execute simultaneously. The opposite holds when running on different cores. Since our experimental machine features simultaneous multi-threading (Intel Hyperthreading), we also considered running on the same physical core, but two different logical cores.

Our results show that, in all configurations, the overhead remains within the measurement error, i.e. in the order of milliseconds. On the application side, storing and restoring the `ymm` registers represent a negligible overhead, consisting in writing/reading a few dozen bytes to/from memory.

Summary. Dynamic binary optimization is a powerful technique to adapt binary programs to the runtime conditions, such as the underlying hardware. The optimization is fully transparent to the user. In addition, since we apply the transformation at runtime, we have the ability to monitor the new performance, and revert to the original code if it is not satisfactory. Finally, modifying binary code requires complex analysis, but we always have to option to abort whenever all conditions are not met.

In the following section, we present a different approach to modifying programs at runtime, based on the functionality of the UNIX loader.

4.3 Function Memoization

This research was developed at Inria within the framework of the PhD of Arjun Suresh. Other participants include Bharath Narasimha Swamy, and André Seznec. Details can be found in the following publication:

- [SSNRS15] Arjun Suresh, Bharath Swamy, Erven Rohou, and André Seznec. Intercepting functions for memoization – a case study using transcendental functions. *ACM TACO*, 2015.

Memoization is the technique of saving result of executions so that future executions can be omitted when the same input set repeats. Memoization has been proposed in previous literature at the instruction level, basic block level and function level using hardware as well as pure software level approaches including changes to programming language.

We proposed [SSNRS15] software memoization for procedural languages like C, Fortran, etc. at the granularity of a function: result of function calls are saved in

a look-up table in memory, so that the result of a future call to the same function with the same parameters comes from the lookup-table rather than from actual execution of the function. This requires that the function be pure: the result is entirely determined by the parameters, and the function has no side-effect. We designed a simple linker based technique for enabling software memoization of any dynamically linked pure function by function interception.

We illustrate our framework using a set of computationally expensive pure functions – the transcendental functions (in the `libm` library).

Our technique does not need the availability of source code and thus can be applied even to commercial applications as well as applications with legacy codes. As far as users are concerned, enabling memoization is as simple as setting an environment variable. Our framework does not make any specific assumptions about the underlying architecture or compiler tool-chains.

Intercepting function calls is implemented by setting the UNIX environment variable `LD_PRELOAD` to the path to our own library. Our own functions implement a software cache. At each invocation, we first check in the cache whether this function has already been called with this parameter. In this case, we directly return the result. Otherwise, we evaluate the function, store the results in the cache and return the result.

Analytical Model. Memoization overhead includes the time needed to evaluate the hash function and the time spent in table lookup. When table entries are located in a cache closer to the processor, less time is spent in reading out the table entries. On a miss, this additional overhead is added to the function execution time. For the hash function we designed, we experimentally measured the time needed for hash table lookups on an Intel Ivy Bridge processor clocked at 2 GHz. Successful table lookup time of around 60 clock cycles (approximately 30 ns) on our benchmark applications. On a miss, we measured the overhead of table overhead[2] to be nearly 110 clock cycles (55 ns) which we attribute to the fact that, during a miss, the table entry is more likely not present in the caches closer to the processor. Both our hit time and miss time compares favorably with the 90–800 clock cycles average execution time of our target transcendental functions.

For a given overhead, we can estimate the potential benefits of memoizing transcendental functions. Let T_f be the execution time of the memoized function, t_h be the time when there is a hit in the memoized table and t_{mo} be the overhead of memoization when there is a miss in the memoized table. We derive an expression for the fraction of calls that should have arguments repeated (H), for memoization to be effective.

$$\begin{aligned} H \times t_h + (1-H)(T_f + t_{mo}) &< T_f \\ H \times t_h + (X + t_{mo}) - H(X + t_{mo}) &< T_f \\ t_{mo} &< H(T_f + t_{mo} - t_h) \\ \implies H &> \frac{t_{mo}}{T_f + t_{mo} - t_h} \end{aligned}$$

[2]Miss overhead adds to the function execution time when there is a miss in the memoization table.

	Function	Hit Time (ns) t_h	Miss Overhead (ns) t_{mo}	Avg. Time (ns) T_f	Repetition Needed H
double	exp	30	55	90	48%
	log			92	47%
	sin			110	41%
	cos			123	37%
	j0			395	13%
	j1			325	16%
	pow			180	27%
	sincos			236	21%
float	expf			60	66%
	logf			62	63%
	sinf			59	65%
	cosf			62	63%
	j0f			182	27%
	j1f			170	28%
	powf			190	26%
	sincosf			63	63%

Table 4.1: Profitability of memoization – 2 GHz Ivy Bridge, GNU `libm`

Table 4.1 summarizes the repetition threshold for the functions of GNU `libm`, on a 2 GHz Ivy Bridge processor. Figure 4.3 plots the profitability curve, i.e. the necessary percentage of repetition of input parameters as a function of the average execution time of the function. We also show where Intel and GNU `libm` stand. The Intel library is more optimized, and thus requires more repetition to make memoization beneficial.

Results. On Intel Ivy Bridge using the GNU compiler, SPEC CPU benchmarks gave benefit ranging between 1 % and 24 % (see Figure 4.4)-(a). Only *bwaves* produces an outstanding and unexpected speed-up of 1.76. The large benefit for mem-

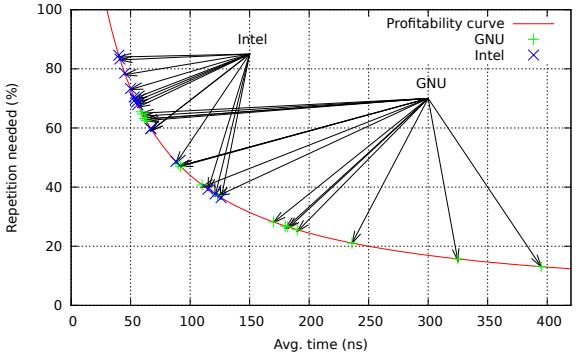


Figure 4.3: Profitability curve of memoizing transcendental functions

oization for *bwaves* is surprising considering that *pow* takes on average around 300 clock cycles only. We found out that this is due to a performance bug in the implementation of *pow* in the GNU `libm` for some inputs[3] which causes a slow down up to 10,000× compared to a normal call. In the case of *bwaves* this happens when computing m^n where m is very close to 1 and n is very close to 0.75. For these input values, the current implementation is found to take more than 1000× the normal execution time. With memoization this long latency is saved and thus we get high benefit. To a smaller extent, this performance bug (in *powf* function) has an impact on the speed-up of *wrf* as well.

Compared to SPEC benchmarks, other applications give much higher benefit for memoization as shown in Figure 4.4-(b). We get a speed-up of 27 % for the ATMI application [MSF+07] (average value for all provided example inputs). This is due to the fact that very expensive Bessel functions – *j0* and *j1* – are used a lot in these programs. *Population dynamics* gives 52 % speed-up as the memoized functions– *log* and *exp* – cover most of the program execution time. *Barsky* improves by 3 % as the memoized function *sin* covers only a part of the program execution. From the Splash 2x benchmarks, *water_spatial* gave 16 %, *fmm* 2 % and *ocean_cp* less than 1 % speed up.

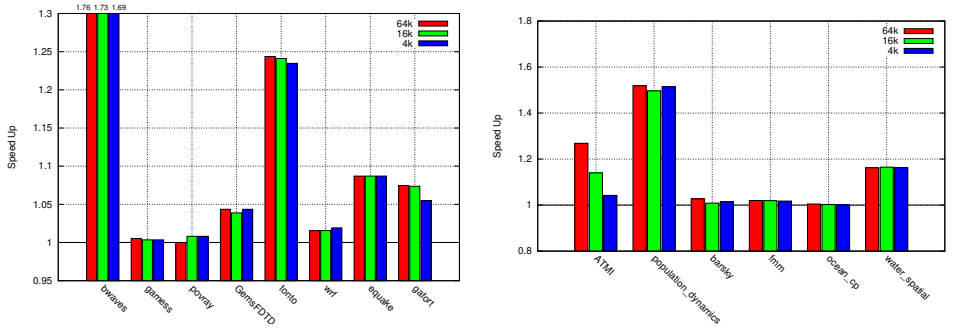
Impact on memory hierarchy. Memoization substitutes execution of code by a table lookup. Given the size of the table, it might interfere with application's data in lower level caches. We measured the L1 and L2 miss rates on all the applications. In most of the applications, the number of L1 misses is reduced by memoization, but by less than 1 %. For *gafort* the L1 miss was increased by 4 % and for *gamess* it was increased by 2 %. For *ATMI* only, increase in L1 miss was abnormally large at 40 %. L2 misses were also reduced for some applications with the maximum being for *barsky* at 2 %. Increase in L2 misses were within 1 % for all applications except *population dynamics*, *gamess*, *povray* and *ATMI*. For *population dynamics* L2 miss increased by 4.6 %, for *gamess* by 3.1 % and for *povray* by 3.6 %. For *ATMI* L2 miss was very high at 46 %. The large cache misses for *ATMI* is expected as it is very critical in Bessel functions (which account for more than 75 % of the execution time) and so with memoization, the table is intensively used causing a lot more cache misses.

Additional results can be found in our TACO publication [SSNRS15], in particular results with a different compiler, on an ARM processor, and the impact of cache associativity.

4.4 Impact and Perspectives

Dynamic binary optimizations potentially have a huge impact: they enable powerful transformations, at the scale of the whole program (i.e. including libraries), without access to source code – thus covering commercial and legacy code, and without the need to restart a long running process.

[3] `https://sourceware.org/bugzilla/show_bug.cgi?id=13932`



(a) SPEC 2006 (b) selected applications

Figure 4.4: Memoization on Intel Ivy Bridge (GNU compiler) for different table sizes

The downside is that tools must be extremely robust. Machine code is not as easy to parse as source code [Val14]. Some instruction sets, such as x86, may not even be statically analyzed, due to their variable length encoding, and potential mix of code and data. Analysis is also complicated by the presence of many irreducible loops, and frequent tail-call elimination in libraries. Correctness is also much more difficult to ensure, as applications are not forced to follow generally accepted ABIs (Application Binary Interfaces). Bruening et al. [BZA12], for example, discuss transparency requirements in mainstream Windows and Linux applications.

Despite the complexity, various projects have shown the power of the approach, for example Transmeta's Crusoe [DGB+03], Transitive's Rosetta, DEC's FX!32 [CHH+98]. IBM Haifa Research Lab has also been working recently on this approach [NED+13].

Our own framework Padrone, discussed in Section 6.4, is still at a prototype stage, and actively being developed. Compared to other systems, we have an advantage: we execute most of the application's code in-place, and optimize only hot functions, a small fraction of the code. And because we "only" re-optimize, we always have the option to abort whenever something cannot be guaranteed correct. Our initial work on re-vectorizing loops to take advantage of the AVX instruction set shows very good results. Ongoing work considers applying more complex loop transformations by translating binary code back to a compiler intermediate representation and leveraging the full power of state-of-the-art static compilation techniques.

Technically much simpler that dynamic binary optimizations, function interception is another way to alter the behavior of a program in binary format. Memoization has shown good hit rates on mathematical functions, and interesting speedups at low cost. Future directions will consider several aspects: 1) assistance of a static compiler to identify pure functions that can be memoized, and possible dedicated hardware to lower the overhead and improve profitability; 2) addition of a helper thread to pre-fill our cache; 3) memoization of non-pure functions; 4) possibility to improve performance at the cost of reduced precision of floating point computa-

tions (compilers already authorize loss of precision, e.g. when the flag `-ffast-math` is passed, in particular in the case of vectorization of reductions). In this case, faster results can be returned by interpolating the values of previous invocations of a function.

Finally, there is an opportunity in coupling the two approaches, dynamic binary analysis providing promising candidates to a memoization framework.

Chapter 5

Interpreters

The previous chapters of this document have looked at various ways to improve the performance of applications *compiled* to native code. The alternative to compilation is interpretation, i.e. the execution of a program, step-by-step, without prior compilation or code generation.

Interpreters go back to the infancy of computer science. At some point, just-in-time (JIT) compilation technology matured enough to deliver much better performance, and was made popular by Java [CFM+97]. Developing a JIT compiler, however, is much more complex and time consuming than an interpreter, and vendors of lower-end devices, where time-to-market is a key factor for commercial success, may decide to content themselves with interpretation.

At the other end of the spectrum, scientists from both CERN and Fermilab report [NC08] that "many of LHC experiments' algorithms are both designed and used in interpreters". The admitted reason is that the software base is so large and complex that changing simulation parameters requires lengthy rebuilds, whereas interpreting is much more straightforward for them. As another example, the need for an interpreter is also one of the three reasons motivating the choice of Jython for the data analysis software of the Herschel Space Observatory [W+04]. Scientists at CERN have also been developing an interpreter for the C/C++ languages for decades [DG87].

More recently, domain scientists started relying on languages such as R, Python, or Matlab, that happen to be mainly executed through interpreters. These languages often feature dynamic characteristics that make JIT compilers inefficient, or at best difficult to design and engineer. For example dynamic type checking requires heavy specialization to achieve decent performance; function overloading requires the ability to unload code at any time, and prevents useful classic optimizations.

Interpreters constitute the class of execution environment of predilection in many situations. They also received renewed attention in the literature. This chapter briefly reviews the factors dominating the performance of interpreters, and then presents our two contributions.

1. We debunk a myth, according to which the performance of naive implementation of interpreters is dominated by the impossibility to predict the target of an indirect branch instruction.

```
1  while (1) {
2   opc = *vpc++;
3   switch(opc) {
4    case ADD:
5     x = pop(stack);
6     y = pop(stack);
7     push(stack,x+y);
8     break;
9
10   case SUB: ...
11  }
12 }
```

(a) C source code

```
1  loop:
2   add    0x2,esi
3   movzwl (esi),edi
4   cmp    0x299,edi
5   ja     <loop>
6   mov    0x..(,edi,4),eax
7   jmp    *eax
8   ...
9   mov    -0x10(ebx),eax
10  add    eax,-0x20(ebx)
11  add    0xfffffff0,ebx
12  jmp    <loop>
```

(b) x86 assembly

Figure 5.1: Main loop of naive interpreter

2. We propose to leverage the vast amount of expertise developed by the community in vectorization to significantly improve the performance of interpreters, thanks to coarse grain *superinstructions*.

The lack of code generation in interpreters requires very different techniques and this constitutes the differentiating factor of the research presented here.

5.1 Performance of Interpreters

The main performance penalty in interpreters arises from instruction dispatch. Figure 5.1-(a) illustrates a basic (naive) implementation of an interpreter: an infinite loop that reads an opcode `opc` from a *virtual program counter* `vpc`, increments the vpc, and jumps to the chunk of code that implements the semantics of the opcode (the *payload*). Many bytecodes also implement an evaluation stack (Java, CLI), but this not a requirement (Dalvik does not).

As a result, each bytecode requires a minimum number of machine instructions to be executed. Figure 5.1-(b) illustrates the assembly code produced by the GCC compiler for the x86 instruction set. In this particular example, executing an ADD bytecode results in ten instructions. This compares to the single native instruction needed for most bytecodes when the bytecode is JIT compiled.

Stack caching. Many bytecodes rely on an evaluation stack to implement computations. The code for `c=a+b` is generated as `load a`, `load b`, `add`, `store c`, and the interpreter typically implements a stack in software. As a result, all accesses to local variables become memory accesses. Conversely, compiled code is likely to promote these values to registers. Stack caching [Ert95] consists in trying to promote the top of the stack (the most frequently accessed elements of the stack) to registers.

As an alternative to stack caching, some virtual machines are register-based. Shi et al. [SCEG08] show they are more efficient when sophisticated translation and optimizations are applied. This is orthogonal to the dispatch loop.

Superinstructions. Sequences of bytecodes are not random, and some pairs are more frequent than others (e.g. a compare instruction is often followed by

a branch). Building superinstructions [PR98] consists in recognizing such sequences of frequently occurring tuples of bytecode and defining new bytecodes, whose payloads are the combination of the payloads of the tuples. The overhead of the dispatch loop is unmodified but the gain comes from a reduced number of iterations of the loop (i.e. one iteration for the equivalent of two bytecodes), hence a reduced average cost. Ertl and Gregg [EG03] discuss static and dynamic superinstructions. We present in Section 5.3 a new technique to generate coarse grain superinstructions thanks to vectorization technology.

Replication. Replication, also proposed by Ertl and Gregg [EG03], consists in generating many opcodes for the same payload, specializing each occurrence, in order to maximize the performance of branch target buffers. The rationale is that each replicated copy will correspond to a different target of the indirect branch, thus exposing more context to the branch predictor. The additional targets of the indirect branch could incur a performance degradation, but it is very limited, and the cost is easily recouped by better prediction. We demonstrate in Section 5.2 that modern predictors no longer need such techniques to capture patterns in interpreted applications.

Jump threading. Jump threading is an optimization of the dispatch loop that consists in bypassing the classical `switch` statement, and jumping directly from one `case` entry to the next. Figure 5.2 illustrates how this can be written (jump threading, though, cannot be implemented in standard C. It is commonly implemented with the GNU extension named *Labels as Values*). The intuition behind the optimization derives from increased branch correlation: first, a single indirect jump with many targets is now replaced by many jumps, reducing aliasing; second, each jump is more likely to capture a repeating sequence, simply because application bytecode has patterns (e.g. a compare is often followed by a jump). Several versions of threading have been proposed: token threading (illustrated in Figure 5.2), direct threading [Bel73], inline threading [PR98], or context threading [BVZB05]. All forms of threading require extensions to ANSI C. Some also require limited forms of dynamic code generation and walk away from portability and ease of development.

We demonstrate in Section 5.2 that the impact of jump threading on modern predictors is much less than it used to be.

5.2 Branch Prediction and Performance of Interpreters – Don't Trust Folklore

This research was developed at Inria. Other participants include Bharath Swamy, and André Seznec. Details can be found in our publication:

- [RNSS15] Erven Rohou, Bharath Narasimha Swamy, and André Seznec. Branch Prediction and the Performance of Interpreters – Don't Trust Folklore. In *CGO*, 2015.

```
void* labels[] = { &&ADD, &&SUB... };
...
goto *labels[*vpc++];

ADD:
   x = pop(stack);
   y = pop(stack);
   push(stack, x+y);
   goto *labels[*vpc++];

SUB:    ...
   goto *labels[*vpc++];
```

Figure 5.2: Token threading, using a GNU extension

As discussed in previous section, conventional wisdom attributes a significant penalty to the indirect branch of the dispatch loop of interpreters, due to its supposedly high misprediction rate. We revisit this assumption, considering current interpreters, and modern predictors. Using both hardware counters and simulation, we show that the accuracy of indirect branch prediction is no longer critical for interpreters. We also compare the characteristics of these interpreters and analyze why the indirect branch is less important than before.

Prediction trends on existing hardware Current versions of Python-3 and Javascript automatically take advantage of threaded code when supported by the compiler. The implementation consists in two versions (plain `switch` and threaded code), one of them being selected at compile time, based on compiler support for the *Labels as Values* extension. Threaded code, though, can be easily disabled though the `configure` script or a `#define`. The current source code of Python-3 also says:

> "At the time of this writing, the threaded code version is up to 15-20% faster than the normal switch version, depending on the compiler and the CPU architecture."

We experimented with Python-3.3.2, both with and without threaded code, and the *Unladen Swallow* benchmarks. Figure 5.3 shows the performance improvement due to threaded code on three successive generations of micro-architectures: Nehalem (2008), Sandy Bridge (2011), and Haswell (2013).

Nehalem shows a few outstanding speedups (in the 30 %–40 % range), as well as Sandy Bridge to a lesser extent, but the average speedups (geomean of individual speedups) for Nehalem, Sandy Bridge, and Haswell are respectively 10.1 %, 4.2 %, and 2.8 % with a few outstanding values for each micro-architecture. The benefits of threaded code clearly decreases with each new generation of micro-architecture.

Intuition The TAGE [SM06] predictor performs very well at predicting the behavior of conditional branches that exhibit repetitive patterns and very long patterns. Typically, when a given (maybe very long) sequence branches before the current program counter was always biased in a direction in the past, then TAGE –

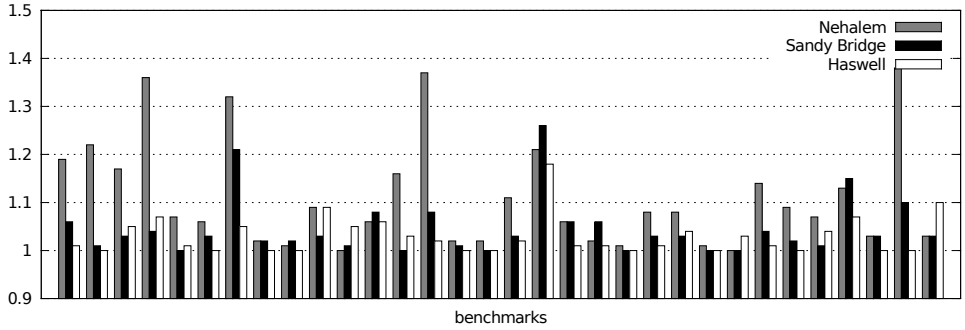


Figure 5.3: Speedup due to threaded code in Python-3

provided it features sufficient number of entries – will correctly predict the branch, independently of the minimum history *le* needed to discriminate between the effective biased path and another path. This minimum path is captured by one of the tables indexed with history longer than *le*. With TAGE, the outcomes of branches correlated with close branches are captured by short history length tables, and the outcomes of branches correlated with very distant branches are captured by long history length tables. This optimizes the application footprint on the predictor. The same applies for indirect branches.

When considering interpreters, the executing program is the interpreter, and the application is the data processed by the interpreter. The executed path is essentially the main loop around the execution of each bytecode. When running on the succession of basic block bytecodes, the execution pattern seen by the `switch` reflects the control path in the interpreted application: in practice, the history of the recent targets of the jump is the history of opcodes. For instance, if this history is `load load add load mul store add` and if this sequence is unique, then the next opcode is also uniquely determined. This history is in some sense a signature of the virtual program counter, it determines the next virtual program counter.

When running interpreters, ITTAGE is able to capture such patterns and even very long patterns spanning over several bytecode basic blocks, i.e. to "predict" the virtual program counter. Branches bytecodes present the particularity to feature several possible successors. However, if the interpreted application is control-flow predictable, the history also captures the control-flow history of the interpreted application. Therefore ITTAGE will even predict correctly the successor of the branch bytecodes.

Experimental Results We experimented with `switch`-based (no threading) interpreters for three different input languages: Javascript SpiderMonkey 1.8.5, Python version 3.3.2, and the Common Language Infrastructure (CLI, aka .NET), and several inputs for each interpreter. Javascript benchmarks consist in Google's *octane*

suite[1] as of Feb 2014, and Mozilla's *kraken*[2]. For Python, we used the *Unladen Swallow Benchmarks*. Finally, we used a subset of *SPEC 2000* (*train* input set) for CLI. All benchmarks are run to completion (including hundreds of hours of CPU for the simulation).

We used GCC4CLI [COR07] (cf. Section 6.6.2) to compile the SPEC 2000 benchmarks. The CLI interpreter is a proprietary virtual machine (cf. PVM in Section 6.6) that executes applications written in the CLI format. Most of standard C is supported by the compiler and interpreter, however a few features are missing, such as UNIX signals, *setjmp*, or some POSIX system calls. This explains why a few benchmarks are missing (namely: `176.gcc`, `253.perlbmk`, `254.gap`, `255.vortex`, `300.twolf`). This is also the reason for not using SPEC 2006: more unsupported C features are used, and neither C++ nor Fortran are supported.

The interpreters are compiled with Intel icc version 13, using flag `-xHost` that targets the highest ISA and processor available on the compilation host machine.

Branch prediction data is collected from the PMU (performance monitoring unit) on actual Nehalem (Xeon W3550 3.07 GHz), Sandy Bridge (Core i7-2620M 2.70 GHz), and Haswell (Core i7-4770 3.40 GHz) architectures running Linux. Both provide counters for cycles, retired instructions, retired branch instructions, and mispredicted branch instructions. We relied on Tiptop [Roh12] (cf. Section 6.2) to collect data from the PMU. Events are collected per process (not machine wide) on an otherwise unloaded workstation.

We also experimented with a state-of-the-art branch predictor from the literature: TAGE and ITTAGE [SM06]. The performance is provided through simulation of traces produced by Pin [L+05]. We used two (TAGE+ITTAGE) configurations. Both have 8 KB TAGE. TAGE1 assumes a 12.62 KB ITTAGE, TAGE2 assumes a 6.31 KB ITTAGE.

Figure 5.4 illustrates the branch misprediction rates of the Python interpreter. Javascript and CLI show similar trends (refer to our publication [RNSS15] for details). The branch misprediction rates are measured in MPKI, misprediction per kilo instructions, generally considered as a quite illustrative metric for branch predictors.

Our measures clearly show that, on Nehalem, on most benchmarks, 12 to 16 MPKI are encountered, that is about 240 to 320 cycles are lost every 1 kilo-instructions. On the next processor generation Sandy Bridge, the misprediction rate is much lower: generally about 4 to 8 MPKI, i.e. decreasing the global penalty to 80 to 160 cycles every Kilo-instructions. On the most recent processor generation Haswell, the misprediction rate further drops to 0.5 to 2 MPKI in most cases, that is a loss of 10 to 40 cycles every kilo-instructions. This rough analysis illustrates that, in the execution time of interpreted applications, total branch misprediction penalty has gone from a major component on Nehalem to only a small fraction on Haswell.

Thanks to simulation, we observed the individual behavior of specific branch instructions. We measured the misprediction ratio of the indirect branch of the

[1] `http://code.google.com/p/octane-benchmark`
[2] `http://krakenbenchmark.mozilla.org/kraken-1.1/`

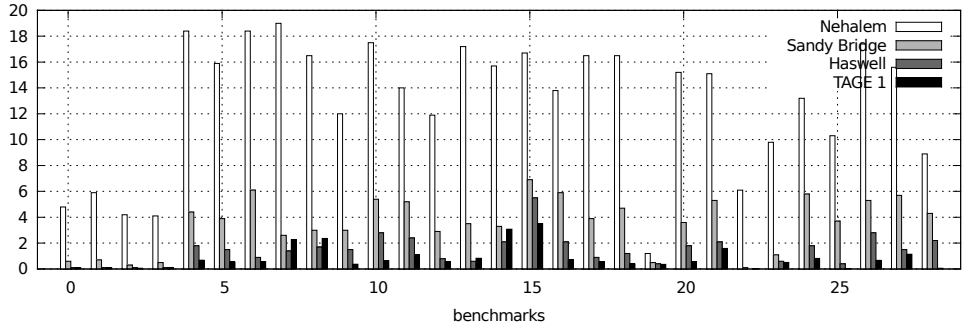


Figure 5.4: Python MPKI for all predictors

dispatch loop in each interpreter. Moreover the source code of Python refers to two "hard to predict" branches. The first is the indirect branch that implements the `switch` statement. The second comes for the macro `HAS_ARG` that checks whether an opcode has an argument. For Python, we also considered this conditional branch.

On Python, the indirect jumps are most often very well predicted for most benchmarks, even by the 6 KB ITTAGE. However, in several cases the prediction of indirect jump is poor (see for example the Python `chaos`, `django-v2`, `formatted-log`, `go`). These cases except `go` are in practice near perfectly predicted by the 12 KB configuration: the footprint of the Python application on the indirect jump predictor is too large for the 6 KB configuration, but fits the 12 KB configuration. `go` needs an even larger predictor as illustrated by the results on the 50 KB configuration. `HAS_ARG` turns out to be very easily predicted by the conditional branch predictor TAGE at the exceptions of the same few outliers with 1 % to 4 % mispredictions.

Summary. Thanks to both simulation and actual measurements, we debunked conventional wisdom, established about a decade ago, about the impact of branch prediction on the performance of interpreter. State-of-the-art predictors are now capable of predicting well the supposedly problematic indirect branch instruction of switch-based interpreters. Still, interpreters suffer several performance bottlenecks, and the next section presents a new approach to superinstructions.

5.3 Vectorization Technology to Improve Interpreters

This research was developed at Inria. Other participants include David Yuste Romero and Kevin Williams. Details can be found in our publication:

- [RWY13] Erven Rohou, Kevin Williams, and David Yuste. Vectorization technology to improve interpreter performance. *ACM TACO*, 9(4):26:1–26:22, 2013.

A common way for improving performance of interpreters is to reduce the number of instructions to dispatch. To this end, common sequences of instructions

are gathered into single superinstructions [EG03, EG04]. The interpreter identifies these repeated patterns and replaces them with the corresponding superinstruction. However, the space of search for superinstructions is limited due to time constraints. Aggressive analysis of the code and powerful transformations are out of reach at runtime.

We proposed [RWY13] to generate coarse-grain superinstructions thanks to vectorization technology. In fact, the first task of vectorizers consists in identifying patterns of repeating instructions applied to consecutive data and to replace them by a single, more "powerful" instruction. The spirit of our approach is similar to regular superinstructions, in the sense that we improve the performance of the interpreter by reducing the number of dispatches. The major differences are as follows. First, the patterns are much coarser grain than traditional superinstructions: they can encompass up to hundreds of dynamic bytecodes. Second, they are not computed by the interpreter, but generated by an offline compiler, and exposed to the interpreter as candidate superinstructions, thanks to predefined builtins. We preserve backward compatibility with legacy interpreters, while enabling very significant speedups on systems based on our proposal.

Our main goal is to deliver performance, and to show that split-vectorization (see Section 3.1) benefits interpreters as well. The efficiency of our proposal relies in the capability of the interpreter to deal with the new builtins. In a JIT compiler, the key feature that delivers performance from vector builtins is the dynamic code generation: each builtin is replaced by its corresponding optimized code sequence. Vector sizes and alignment constraints are materialized, and standard optimizations (such as constant propagation, dead code elimination, function inlining) remove inefficiencies. Interpreters lack this code generation capability.

Vector Superinstructions To handle the builtin, we extended the IR instruction set of our interpreter with 121 new SIMD IR instructions, matching vector operations over different data types, and some primitives required for the target abstraction. During the construction of the IR representation, the bytecode verifier recognizes the calls to the builtins and constructs SIMD IR instructions instead.

The new vector superinstructions are similar to their scalar counterparts, but their payload is much coarser grain. Figure 5.5 illustrates this in the case of the addition. Most of the arithmetic and data manipulation instructions are implemented as loops performing the requested operations over all elements of the input vectors. The improvement relies on the fact that these loops are statically compiled within the interpreter, as opposed to being in the application bytecode. Interpretation of entire vector idioms is skipped, the number of dispatched bytecodes is reduced by a factor proportional to the vectorization factor.

Mapping to actual SIMD instructions On top of reducing the number of dispatches, we also analyzed the impact of providing an implementation of SIMD IR instructions accelerated with the *native* SIMD instruction set. Many of the SIMD IR instructions have a straightforward translation on typical vector instructions, and many more can be written by combining a few of them.

```
while (1) {
 opc = *vpc++;
 switch(opc) {
  case ADD:
  x = pop(stack);
  y = pop(stack);
  push(stack,x+y);
  break;
 }
}
```

(a) Scalar ADD

```
while (1) {
 opc = *vpc++;
 switch(opc) {
  case ADD:
    ...
  case ADD_VEC_BYTE:
  vx = pop(stack);
  vy = pop(stack);
  for(i=0; i < 16; i++)
    vz[i] = vx[i] + vy[i];
  push(stack, vz);
  break;
 }
}
```

(b) Vector ADD (on bytes)

Figure 5.5: Implementation of Vector Superinstruction

```
1  while (1) {
2    opc = *vpc++;
3    switch(opc) {
4      case VEC_ADD_BYTE:
5      vx = pop(stack);
6      vy = pop(stack);
7      vz[i] = __builtin_ia32_paddb128(vx, vy);
8      push(stack, vz);
9      break;
10   }
11 }
```

Figure 5.6: Mapping a vector superinstruction to native x86 SSE

We implemented this optimization of the *interpreter code* thanks to a GCC GNU extension that provides both vector types and operators. Simple arithmetic is automatically detected by GCC: a simple `a+b` where `a` and `b` are of type `v4sf` (i.e., a vector of four floats) generates a single `addps` SSE machine instruction. More advanced computations, such as dot product, interleaving, shuffling, and so on, require extensions, also provided by GCC. Figure 5.6 illustrates the code of the interpreter when SIMD IR instructions are accelerated (line 7).

The performance gain of this optimization is second order compared to the effect of dispatch reduction as consequence of the SIMD IR instructions. The reason is the following: the percentage of optimized code in the whole interpreter is relatively small in comparison with the overhead of the dispatch mechanism. Only the implementation of IR instructions in the `switch` entries are accelerated, and only when hardware support is available. Still, for loop intensive applications, the speedup is significant.

Experimental Results The intermediate representation and the offline compiler reuse unmodified previous work on split-vectorization (Section 3.1): base CLI bytecode, vector builtins, and the GCC4CLI compiler. The interpreter is PVM, as in our previous work on interpreters (Section 5.2). We targeted two instruction sets: the various versions of x86's SSE and the ARM NEON. We experimented with the

Polybench 1.0 suite [PBB+10], and we also added some kernels from our previous work on vectorization for JIT compilers [NDR+11]. These benchmarks are loop kernels, performing computations on arrays. The data type of the array elements is parametric. For each benchmark, we varied the type of the main arrays among `char`, `short`, `int`, `float`, and `double` (only `double` and `float` are reported in this document, refer to our publication [RWY13] for all results).

Figure 5.7 illustrates the speedups brought by vector superinstructions, for ARM (left bar, white) and x86 (central bar, gray). In addition, the reduction in the number of IR bytecode dispatches is shown in the rightmost bar (black). Note that this reduction is independent of the underlying architecture as long as the same interpreter runs the bytecode.

Average speedups range from 9 % to 8× across all the data type configurations, in both evaluated architectures. Our experiments show that the performance increase is directly correlated with the reduction in the number of dispatched IR bytecodes. Also, the results show that data types are influential in the observed speedups. On x86, kernels operating on `char`, `short` and `integer`/`float` benefit from an average 8×, 4× and 2× speedup respectively. The `double` data type results in 9 % speedup. Similar numbers are observed for ARM. The reason is as follows: the amount of overhead removed with the introduction of SIMD IR instructions is mostly proportional to the vectorization factor. Some unavoidable overhead remains, inherent to the vectorization technique, explaining why the speedup is not strictly equal to the vectorization factor. This overhead arises from additional code to handle alignment on iteration spaces that are not multiple of the vectorization factor.

Across all data types, the performance of `jacobi` is almost unchanged. The reason is that the only vectorized loop is an inner loop that copies an array into another one. The benefits of this optimization, while theoretically non null, is hidden by the weight of the rest of the computation. Similarly, the kernels `correlation.double` and `covariance.double` experience slowdowns. In these two cases, the small benefit of the new IR instructions is insufficient to recoup the overhead of vectorization.

Combining the dispatch benefits of SIMD IR instructions with the exploitation of the native SIMD instruction set yields extra performance gains. In this scenario, SIMD IR instructions are executed by means of one or few native instructions. Figure 5.8 illustrates, for each kernel, the speedups obtained by SSE4.1 and NEON. The kernels are distributed along the horizontal axis. We first observe that the trends are similar for the two instruction sets: SSE and NEON can accelerate the same parts of the interpreter. SSE achieves a better speedup: 58 % on average (42 % when ignoring the four outstanding values), compared to NEON's 22 %. Two reasons explain the lower performance of NEON:

1. some SIMD primitives could not easily be expressed with the NEON instruction set, hence are scalarized;

2. NEON operates on smaller vectors (64 bits), hence compared to SSE we need twice as many instructions to implement the 128-bit vector semantics[3].

[3]NEON can operate on both 64- and 128-bit vectors. We chose to experiment with 64-bit to differentiate from SSE's 128-bit.

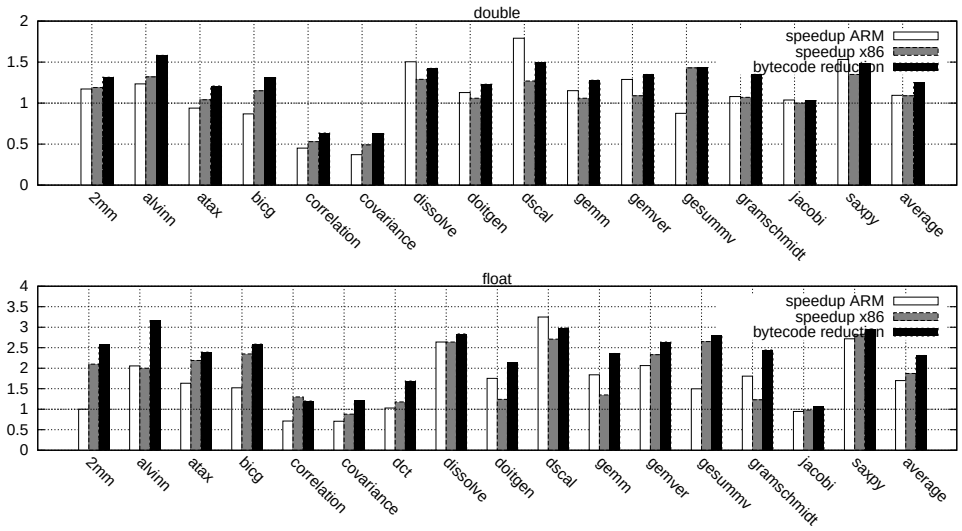


Figure 5.7: Speedup and reduction in number of executed bytecodes

Four benchmarks have an outstanding performance with SSE, with a speedup above 3×: `gramschmidt` for data type `double` and `float`, `doitgen float` and `gemm float`. These numbers are due to a particularly under-performing reference.

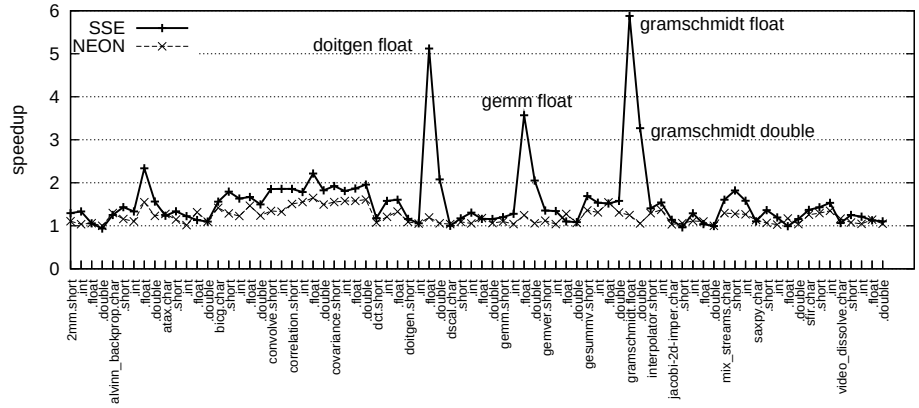


Figure 5.8: Speedup when using native SIMD instruction set

Summary. Split-vectorization, proposed in Section 3.1, delivers performance thanks to the capability of the JIT compiler to optimize and generate code. We showed that interpreters can still leverage the same speculatively vectorized bytecode, by processing the builtins within the interpreter's dispatch loop, in a way similar to superinstructions, but at a much larger granularity.

5.4 Impact and Perspectives

Programs can be executed in two ways: compiled to native code or interpreted depending on the capability to generate binary code (with many hybrid solutions between these two extremes). Interpreters are slower, typically by at least an order of magnitude, but they are much easier to develop and retarget to new devices. To overcome the perceived overheads of interpreters, the community has proposed many techniques. And while some are still valid, a few of them are based on assumptions about the hardware made a decade ago, and just no longer true today. We hope that we have debunked a well installed myth about the predictability of indirect branches in interpreters, and that designers and developers will focus on code readability and maintainability instead of obscure code transformations to address obsolete characteristics of the hardware.

We have also shown that vectorization can dramatically improve performance of selected loops and applications. Regular applications (many scientific applications fall in this category) are likely to feature such vectorizable loops, hence many domain scientists could potentially benefit from such a technique.

More generally, general purpose processors are designed – by definition – to be efficient on a large class of applications. However, when applications are interpreted, the only running application is the interpreter (and the user's application is the data of the interpreter). In this context, many decisions could be reconsidered: for example, the instruction cache only needs to fit the interpreter, while the application and its data share the data cache. Without stack caching, many register accesses turn into memory accesses. Future work will investigate how program characteristics vary in the presence of an interpreter.

Chapter 6

Design and Development of Infrastructures and Tools

All the work presented in the previous chapters of this document required extensive developments for proper assessment and validation. As usual, research contributions must build on top of the state-of-the-art. In the case of compilers, they must compare with existing industrial-quality products. In favorable cases, an optimization can be inserted in the sequence of passes of an existing academic research prototype. In many cases, however, significant development is required. Building an entire optimizing compiler for a language such as C (not to mention C++) is out of reach of an academic effort: compilers now consist of millions of lines of code. The situation is even worse in the case of virtualization and split-compilation where two full-scale compilers (static and dynamic) must be developed and synchronized. Consequently, research proposals must usually be integrated inside existing compilation frameworks, such as GCC or LLVM. This limits the development effort, but makes for a steep learning curve.

Proper validation also requires extensive testing, using commonly accepted benchmarks (such as the SPEC CPU suite [Hen06] in our community, but also collections of real applications), and relevant input data.

In this chapter, we describe our most significant developments. Some of these tools are registered at the French *Agence de Protection des Programmes* (APP). When available, we provide the registration number. We first describes Salto (Section 6.1), a system for assembly language transformation and optimization, that serves as an post-pass optimizing backend, run after the compilation steps. We then present Tiptop (Section 6.2), a standalone Linux utility designed to facilitate the understanding of low level micro-architectural events. If-memo is described in Section 6.3. It is a tool that lets users intercept pure functions (such as the transcendental functions of `libm`), caching the results to avoid future invocations. Section 6.4 introduces our latest academic software contribution: Padrone, a library and API designed to help programmers monitor and optimize executables while they run. Benchmarks play an important role in assessing the quality of scientific results. Section 6.5 describes our contributions in that area. In Section 6.6, we briefly present industrial developments. We conclude in Section 6.7.

6.1 Salto – Optimizing Assembly Language

Developed at IRISA, 1996–1998, within the European FP5 Project OCEANS. Other participants include François Bodin and André Seznec. 60k lines of C++. Open-source: `https://team.inria.fr/alf/software/salto/`. APP IDDN.FR.001.070004.00.R.C.1998.000.10600.

As part of my PhD work, I designed Salto (System for Assembly Language Transformation and Optimization) [RBS+96]. It is a retargetable framework to develop all the tools that are needed for performance tuning on low-level codes (optimizers as well as profilers).

Salto consumes and produces assembly code (cf. Figure 6.1). As such, it is language- and compiler-independent and is easily integrated in build environments. In a compilation environment, such a tool avoids the burden of maintaining a compiler front-end and back-end. Compared to working directly on native binary code, assembly is still relatively readable to facilitate debug and maintenance.

Salto provides a well-defined API to let clients iterate – in a target-independent manner – over control-flow graphs, basic blocks and instructions. Resources used at each pipeline stage at are available. Many architectures have been described, including Sparc, ARM, MIPS, Cray, a subset of x86, Philips TM1000. It has been instrumental to the OCEANS project, and in particular it is a key ingredient of GCDS (see Section 2.2) and iterative compilation (see Section 2.3), as well as our retargetable framework for software pipelining [BCE+98].

CaaS Finally, thanks to Salto, we explored the feasibility of remote distributed compilation as early as 1998 – a concept maybe better named today Compilation as a Service (CaaS). With multiple partners contributing to the compiler, it proved easier to run the optimizers as servers and to move the code under compilation than to deploy several instances of each component at all partners sites (running different OS, tools, etc.) Our system ran as a web service (before the name was even coined) between IRISA, INRIA Rocquencourt, the universities of Leiden (The Netherlands) and Manchester (UK).

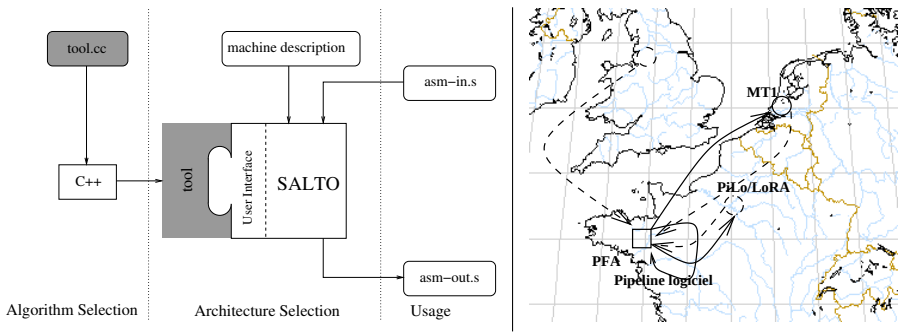


Figure 6.1: Salto: high level description and integration in OCEANS

6.2 Tiptop – Monitoring Hardware Counters

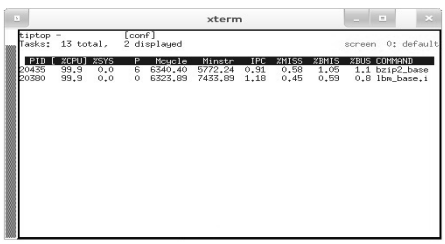
Tiptop is being developed at Inria since 2011. 8k lines of C. Open-source: `http://tiptop.gforge.inria.fr`. APP IDDN.FR.001.450006.000.S.P.2011.000.10800.

Moore's law drives the complexity of processor micro-architectures, which impacts all other layers: hypervisors, operating systems, compilers and applications follow similar trends. While a small category of experts is able to comprehend (parts of) the behavior of the system, the vast majority of users are only exposed to — and interested in — the bottom line: how fast their applications are running. UNIX users typically rely on commands such as *ps* or *top* and look at the column `%CPU` for their processes. When this number is significantly below 100 %, they can investigate the reasons: resource conflicts (e.g. more processes than hardware threads), slow I/O, virtual memory effects, etc. When the CPU usage is close to 100 %, users can only conclude that there is no visible reason to be concerned. CPU usage, however, only tells one part of the story: how often processes are scheduled for execution by the operating system. It says nothing about the way execution proceeds.

We propose [Roh12] to take advantage of hardware performance counters to expose some details of the execution of applications that are currently not easily available to the average user. *Tiptop* is a new tool that is as easy to use as *top*. It requires neither special privilege, nor application source code, nor expert knowledge whatsoever. Simple metrics can let users feel how fast their applications are actually running. Advanced users can also use it to compute more sophisticated ratios and get deeper insights, while still using the same simple tool. Expressions to compute can be defined in an external XML file.

Tiptop is a command-line tool for Linux, purposely very similar to the popular *top* utility. Two running modes provide for different needs. The *live mode* periodically refreshes the screen with new values of the monitored events (similar to *top*) and lets users interactively inspect processes. The *batch mode* produces the same information, but as a streaming text output, similar to *top -b*, convenient for further processing. Figure 6.2 illustrates its output at a glance.

Tiptop supports several architectures: x86, ARM, PowerPC, Sparc.



Live-mode

PID	USER	%CPU	Mcycle	Minst	IPC	DMIS	COMMAND
2962	user1	100.0	26456	52125	1.97	0.0	process1
22831	user3	100.0	26417	34996	1.32	0.0	process2
2954	user1	99.9	28180	63941	2.27	0.0	process3
2969	user1	99.9	28184	66409	2.36	0.0	process4
22833	user3	99.9	26419	30844	1.17	0.0	process5
25242	user2	99.9	28187	18736	0.66	0.9	process6
2944	user1	99.8	26424	45582	1.73	0.0	process7
2965	user1	99.8	28091	40386	1.44	0.0	process8

Batch-mode

Figure 6.2: Tiptop snapshots

6.3 If-memo – Function Memoization

If-memo is being developed at Inria since 2012 within the PhD of Arjun Suresh. It consists in 3k lines of C and assembly snippets. Other participants include Bharath Swamy. APP IDDN.FR.001.250013.000.S.P.2015.000.10800

Memoization is the technique of saving result of executions so that future executions can be omitted when the inputs repeat. Memoization has been proposed in previous literature at the instruction level, basic block level and function level using hardware as well as pure software level approaches including changes to programming language.

We proposed software memoization of pure functions for procedural languages (see Section 4.3). We rely on the operating system loader, taking advantage of the `LD_PRELOAD` feature of UNIX systems. By setting this variable to the path of a shared library, we instruct the loader to first look to missing symbols in that library. Our library redefines the functions we wish to intercept.

The interception code is very straightforward (see also the algorithm on Figure 6.3): it receives the same parameter as the target function and checks in a table (a software cache) if this value is readily available. In the favorable case, the result value is immediately returned. Otherwise, we invoke the original function, and store the result in the cache before returning it.

The table is of limited size and it is implemented as a cache. Older values may be evicted by newer values. The input parameter serves as a tag to guarantee the validity of the accessed data. The table is indexed through a simple yet efficient hash function: we repeatedly XOR the most and least significant bits of the parameter value until we reach the necessary bit-width. For parameters of type `double` (64 bits) and a 64k-entry table, two XOR operations are enough. For smaller tables, we simply mask the higher bits.

Our technique does not require the availability of source code and thus can be applied even to commercial applications as well as applications with legacy codes. As far as users are concerned, enabling memoization is as simple as setting an environment variable. We validated If-memo with x86-64 platform using both GCC and icc compiler tool-chains, and ARM cortex-A9 platform using GCC.

```
double sin(double x)
{
  idx = hash(x);
  if (table[idx].tag) != x) {
    table[idx].tag = x;
    table[idx].val = real_sin(x);
  }
  return table[idx].val;
}
```

Figure 6.3: Interception pseudo-code, example for function *sin*

6.4 Padrone – Dynamic Binary Rewriter

Padrone is being developed at Inria since 2012, partly within the framework of the Nano2017 collaborative project, and the PhD of Nabil Hallou. The main developer is Emmanuel Riou. Other participants include Alain Ketterlin and Philippe Clauss. Padrone is 9k lines of C. APP IDDN.FR.001.460025.000.S.P.2014.000.10700.

As discussed in Chapter 4, many programs are under-optimized for the hardware on which they run. To support our research, we proposed Padrone [RRC+14], a new platform for dynamic binary analysis and optimization. It provides an API to help clients design and develop analysis and optimization tools for binary executables. Padrone attaches to running applications, only needing the executable binary in memory. No source code or debug information is needed. No application restart is needed either. This is specially interesting for legacy or commercial applications, but also in the context of cloud deployment, where actual hardware is unknown, and other applications competing for hardware resources can vary. Profiling overhead is minimal.

Padrone interacts with the target process through Linux features, such as the `ptrace` system call, the `/proc` virtual filesystem. It also takes advantage of the hardware performance monitoring unit. As shown in Figure 6.4, Padrone executes in a different address space from the target process. This implies minimal impact on the target process, and favors transparency [BZA12]. In many scenarios, in the interest of performance, we also execute most of the target code in-place, and only send optimized code in the code cache. This is in contrast with existing solutions such as Pin [L+05] or DynamoRIO [Bru04].

We illustrated it through two examples [RRC+14]. First, we show how we measure the performance of the hotspot of benchmarks. Second, we replace the hotspot of a function by an optimized version, while the program runs.

We believe Padrone fits an empty design point in the ecosystem of dynamic binary tools.

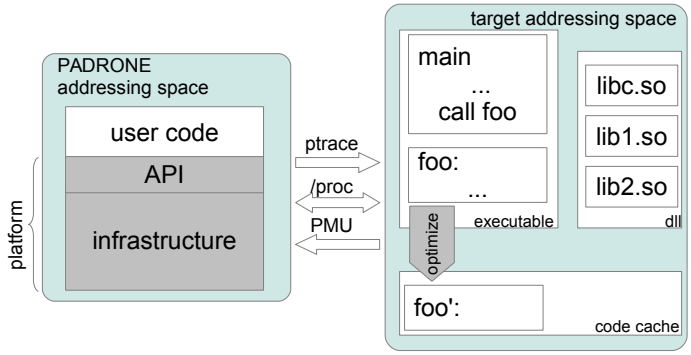


Figure 6.4: Padrone

6.5 Benchmarks

This work spans a decade. The effort on MiDataSets started at STMicroelectronics as a collaboration with Grigori Fursin from Inria Saclay in 2005. The rest of this work was done at Inria. Thierry Lafage contributed to the the MoBS publication [RL10] in 2009–2010, within the context of the Nano2012 program. Finally, my partial involvement with the ParaSuite benchmarks dates from 2013, led by Sylvain Collange and Thibault Person.

As in all other scientific disciplines, our results must be validated by experiments, which often consist in running benchmarks to confirm our claims.

Application benchmarking is a widely trusted method of performance evaluation. Compiler developers rely on them to assess the correctness and performance of their optimizations; computer vendors use them to compare their respective machines; processor architects run them to tune innovative features, and – to a lesser extent – to validate their correctness. Benchmarks must reflect actual workloads of interest, and return a synthetic measure of "performance". Often, benchmarks are simply a collection of real-world applications run as black boxes.

Real applications, however, are often developed for a specific family of processors or operating systems. Older applications also rely on old programming styles or language standards. In collaboration with Grigori Fursin from Inria Saclay, we spent a significant effort porting the MiDataSets suite [FCOT07] to a level where these programs can be compiled and run on many targets, and their outputs validated against reference file. The outcome of this effort is publicly available on the SourceForge repository [1]

Using this popular suite of benchmarks, we also identified a number of pitfalls [RL10] that derive from using applications as benchmarks. In particular, we advocate the fact that correctness should be defined by an expert of the application domain, and the test should be integrated in the benchmark.

With the advent of multi- and many-core processors, as well as parallel accelerators, most execution platforms are now parallel. Research in architecture, compilers and systems focus on these parallel targets, and proposals must be validated preferably with parallel applications that are representative of the current or future workloads. We initiated the project ParaSuite `https://parasuite.inria.fr` that aims at taking advantage of the wide spectrum of parallel applications developed within Inria to provide a modern suite of parallel benchmarks to the community.

[1] `http://sourceforge.net/projects/cbenchmark/files/cBench/V1.1/`

6.6 Industrial Developments

This section describes developments conducted at STMicroelectronics. Due to their industrial context, the effort dedicated to dissemination and publication was limited.

6.6.1 LxBE – VLIW Compilation

> This work was performed at STMicroelectronics in 2000–2004. Other participants include a team of eight engineers and PhDs. LxBE is 300k lines of C++. It is proprietary code.

VLIW processors do not have any hardware to detect dependencies between operations, or any out-of-order capability. They are easier to design, cheaper to manufacture, and they consume less power. But the performance comes entirely from the compiler's ability to extract instruction level parallelism.

LxBE stands for Lx back-end. It is the codename of a effort to produce a state-of-the-art, industrial-quality, and retargetable compiler for the ST200 processor family. It was designed to be shipped to real customers, implying a significant effort of documentation, generating proper error messages, etc. At the same time, it was also flexible enough to be used as a research vehicle.

The back-end was implemented from scratch, in an effort initiated by Josh Fisher's group at Hewlett-Packard Laboratories, soon joined by the AST (Advanced System Technology) unit of STMicroelectronics. We connected it with a compiler front-end developed by the Portland Group Inc. (PGI), one of the leaders of compilers and tools for parallel computing, and a subsidiary of STMicroelectronics at that time. Connecting two large pieces of software independently developed proved to be a large and complex effort.

Robustness was a major concern. Benchmarking and testing were a significant part of the global effort, as well as generation of proper debug information, and clean integration in the overall toolchain. In spite of the industrial focus, we obtained and published new results about alias analysis [GCCRR02, CGRCR04] (Section 2.1).

6.6.2 Processor Virtualization

> This work was performed at ST200, with a team of engineers and PhDs. GCC4CLI consists in 40k lines of C added to the GCC middle-end. It is open-source: `https://gcc.gnu.org/projects/cli.html`. PVM consists in 170k lines of C. It remained proprietary.

As an alternative to the burden of deploying statically compiled code, we started exploring more dynamic aspects of compilation: JIT compiler and virtual machines for embedded systems. Introducing a platform-neutral program representation in the form of a bytecode is a major issue for application portability. In the embedded system industry, the execution targets evolve very rapidly and are often heterogeneous. JIT compilers and virtual machines smooth the development flow and make the integration easier; they also make it possible to postpone the allocation of code fragments to the different processors, even until run-time [ROÖC10].

For pragmatic reasons, we adopted the well defined and extensively documented CLI format (also known as the core of Microsoft .NET). We developed both the bytecode producer as an extension of the GCC compiler (GCC4CLI), and bytecode consumers: an interpreter and JIT compilers.

We showed that the bytecode is typically smaller than the same application written in native code, even for dense instruction sets like ARM Thumb or SH-4 [CR05]. Code size is extremely important in the embedded world because it directly translates into the cost of the flash memory. We also showed that applications compiled to bytecode are not necessarily slow, and that the average performance is not very different from programs compiled directly to native code [CCFP+08], provided that a reasonable set of optimizations is available in the JIT compiler. This result debunked a myth that using bytecodes results in a "loss" of information that, in turn, degrades performance.

GCC4CLI Most legacy code for embedded systems is written in the C language. Because there was no C compiler generating CLI, we developed a back-end for GCC [COR07]. Named GCC4CLI, this compiler is more than a mere port of GCC: CLI is a strongly typed representation, forcing us to maintain high-level information. The evaluation stack – another common feature of virtual machines – also requires special treatment. GCC4CLI adds a new intermediate representation to the GCC compiler, along with a dozen new optimizations specific to the CLI format [SOR09]. Good quality code required developing stack-oriented optimizations, such as elimination of redundant stack accesses [COR07]. GCC4CLI was instrumental to our split-compilation research, in particular the generation of vectorized bytecode [Roh10] for our Vapor SIMD [RDN+11, NDR+11] (see Section 3.1) and interpreter superinstructions (see Section 5.3).

Beyond the need to handle the stack, compiling the C language to the CLI representation proved to be a challenging task, due to the fact that CLI is stricter than C in several aspects (management of data types, function prototypes, switch statements, *vararg* functions...) We studied the relations of the various components of the C ecosystem (language standard, ELF format for executables and libraries, operating system interface) and we proposed solutions for efficient compilation of C to CLI [ROC10].

PVM We designed and built PVM, an entire virtual machine for ARM- and ST200-based STMicroelectronics platforms, featuring an interpreter and JIT compilers, capable of mixed-mode execution (switching between interpretation and JIT-compilation). We demonstrated that JIT technology is a viable approach even for real-time embedded systems.

The operations of the virtual machine are under the control of the VES (Virtual Execution System). The VES first requests the loader to locate the executable and to store it in memory. The verifier is then invoked to check the legality of the bytecode, as mandated by the CLI standard. In particular, it checks that local variables are only assigned values of the proper type, and that no stack overflow or underflow can occur. While processing the CLI input stream, it emits an internal representa-

tion (IR bytecode) that is used by the rest of virtual machine. IR resembles CLI, but it is more efficiently processed by the interpreter. For example, polymorphic CLI operators such as add are replaced by typed operators like add.i4 (32-bit integer addition) or add.r8 (double precision floating point). When the interpreter is invoked, frequent bytecode patterns are combined into pre-built superinstructions [EG03], to reduce the number of dispatched bytecodes and improve performance.

PVM's interpreter was instrumental to our research in applying vectorization technology to interpreters (cf. Section 5.3), as well as debunking the folklore about the performance of indirect branch prediction in interpreters (cf. Section 5.2).

6.7 Impact

Software development is undoubtedly a major component of computer science. We try to summarize the advent of our main developments.

Salto. Salto is released to other research groups upon request. It has been used by for diverse purposes: dependence analysis [RBES00], computation of program slices [LS00], application specific processors [CM99] or computation of worst-case execution time [CP01]. Almost 20 years after its initial development, PhD students, in our group and outside, were still relying on it for their research [Les13]. Salto has also been used for teaching back-end optimizations at the University of Rennes 1. It is in use in the WCET toolchain Heptane developed in the Inria ALF project-team.

A new version has been engineered by the French start-up company CAPS Entreprise. Hundt et al. [HRTV11] also re-implemented the same idea in a tool called MAO, that reads assembly and produces optimized object files.

Tiptop. Tiptop has been presented at the 4th meeting of the French community in compilation[2] in Saint-Hippolyte, France. It has been adopted for teaching purposes, at the University of Lille and at ENS Lyon. It is also a key ingredient of the lab session of the 2014 summer school *École Jeunes Chercheurs en Programmation* organized in Rennes. Tiptop contributed to the research in architecture of colleagues of the ALF group [NSS13] as well as other research at IRISA/Inria where performance is key (see the PhD thesis of Gylfi Đór Guðmundsson on parallelism and distribution of very large scale content-based image retrieval [Guð13]). Non computer scientists (biologists at INRA, working on the population dynamics of aphids [CPM+14]) found it useful to diagnose performance bottlenecks.

In July 2014, tiptop was integrated in major Linux distributions: Debian, Ubuntu, and Fedora.

Padrone. Padrone is not publicly available, but distributed under a license agreement. It has been transferred to Télécom Bretagne for research purpose on dynamic software updates. It was also the framework of choice for our work on dynamic reoptimization of vectorized code (Section 4.2). Padrone is instrumental to three

[2] `http://compilation.gforge.inria.fr/2011_12_SaintHippolyte/Slides/erven.pdf`

PhDs in the Inria ALF project-team (Nabil Hallou, Arjun Suresh and Arif Ali Ana-Pparakkal). Further work will extend the platform for more aggressive optimizations.

Benchmarks. MiDataSets have been downloaded nearly 4000 times (according to SourceForge statistics). They have been used in various articles, and are instrumental to the cTuning project [3].

The ParaSuite initiative raised interest from colleagues at Inria. At the time of writing, eight benchmarks have been integrated and released.

In our future work, we consider focusing on benchmarks for WCET, such as the Mälardalen suite [GBEL10], and to extend them with more input sets, thus exploring more program paths and execution times.

Industrial Developments. LxBE was a proprietary development, as such it will not be publicly released. Still, it proved useful to transfer ideas to product groups within STMicroelectronics.

GCC4CLI has been contributed to the community and it is publicly available in the usual GCC repository. This compiler also served as a base for previous collaborations with Politecnico di Milano (Italy), IBM Haifa Research Labs (Israel), Harvard University (USA), INRIA Saclay and STMicroelectronics Grenoble. I am still a maintainer of this branch of GCC.

PVM, developed at STMicroelectronics, was also further used in several lines of research: our work on generating superinstructions from vectorization technology [RWY13] (see Section 5.3), and our recent study of the behavior of indirect branch instructions [RNSS15] (see Section 5.2).

Salto

[RBS+96] Erven Rohou, François Bodin, André Seznec, Gwendal Le Fol, François Charot, and Frédéric Raimbault. Salto: System for assembly language transformation and optimization. In *Workshop on Compilers for Parallel Computers (CPC)*, Forschungszentrum Jülich GmbH, 1996.

[BCE+98] François Bodin, Zbigniew Chamski, Christine Eisenbeis, Sylvain Lelait, Erven Rohou, Antoine Sawaya, André Seznec, and Jian Wang. Towards a retargetable framework for software pipelining. In *International Workshop on Compilers for Parallel Computers (CPC)*, 1998.

Tiptop

[Roh12] Erven Rohou. Tiptop: Hardware performance counters for the masses. *ICPP Workshops*, 2012.

If-memo

[3] `http://ctuning.org`

[SSNRS15] Arjun Suresh, Bharath Swamy, Erven Rohou, and André Seznec. Intercepting functions for memoization – a case study using transcendental functions. *ACM TACO*, 2015.

Padrone

[RRC+14] Emmanuel Riou, Erven Rohou, Philippe Clauss, Nabil Hallou, and Alain Ketterlin. PADRONE: a Platform for Online Profiling, Analysis, and Optimization. In *Dynamic Compilation Everywhere*, 2014.

Benchmarks

[RL10] Erven Rohou and Thierry Lafage. The pitfalls of benchmarking with applications. In *Workshop on Modeling, Benchmarking and Simulation*, 2010.

Industrial Developments

[GCCRR02] Marco Garatti, Roberto Costa, Stefano Crespi Reghizzi, and Erven Rohou. The impact of alias analysis on VLIW scheduling. In *4th International Symposium on High Performance Computing (ISHPC)*, 2002. LNCS 2327.

[CGRCR04] Roberto Costa, Marco Garatti, Erven Rohou, and Stefano Crespi Reghizzi. Hardware parameters of VLIW cores and code quality factors affecting alias analysis impact. *ST Journal of Research*, 1(2):97–108, 2004.

[ROÖC10] Erven Rohou, Andrea C. Ornstein, Ali Erdem Özcan, and Marco Cornero. Combining processor virtualization and component-based engineering in C for many-core heterogeneous embedded MP-SoCs. In *Workshop on Programming Models for Emerging Architectures (PMEA)*, 2010.

[CCFP+08] Marco Cornero, Roberto Costa, Ricardo Fernández Pascual, Andrea C. Ornstein, and Erven Rohou. An experimental environment validating the suitability of CLI as an effective deployment format for embedded systems. In *HiPEAC*, 2008. LNCS 4917.

[SOR09] Gabriele Svelto, Andrea Ornstein, and Erven Rohou. A stack-based internal representation for GCC. In *International Workshop on GCC Research Opportunities (GROW), in conjunction with HiPEAC*, 2009.

[COR07] Roberto Costa, Andrea C. Ornstein, and Erven Rohou. CLI back-end in GCC. In *GCC Developers' Summit*, 2007.

Chapter 7

Conclusion and Perspectives

In the last two decades, compilation technology has evolved in many directions. Performance and code size used to be the main drivers. Many new metrics have been added: energy, power, temperature, power efficiency... Even performance can be defined in many different ways: minimum latency, maximum throughput, worst case execution time. In the same period of time, architecture have also become incredibly complex, in terms of underlying micro-architecture, but also architecture: multicores and manycores are now commonplace, heterogeneous platforms enter the general purpose market (IBM's Cell, Intel's Xeon Phi, AMD's Fusion).

Recently, many scientific fields started relying heavily on computational resources: physics, biology, medicine, but also social sciences gather huge amounts of data, and require considerable amount of processing time to analyze them. These scientists are not necessarily computer scientists. We observe a growing discrepancy between, on the one hand, the complexity of the workloads and the computing systems they have access to, and on the other hand, the expertise to optimize and to map efficiently computations to compute nodes.

The compilation community had to adapt to all these evolutions. To make things worse, the clock frequency stopped increasing automatically (magically?) about a decade ago, putting more pressure on the software stack to deliver the expected performance increase. Various techniques have emerged, including iterative compilation, split-compilation.

Dark silicon (the fact that manufacturers can integrate many transistors on a die, but only a fraction can operate at any given time due to power constraints) advocates for heterogeneous chips by design. Process variation and business constraints also force vendors to try to sell chips, even if not 100 % correct, possibly at a lower price (a technique called *binning*). This results in *de facto* heterogeneity. Compilers of the future will face an increasing diversity of targets. The anticipated end of Moore's Law will also probably generate new evolutions in the computing systems landscape.

Embedded systems have followed the evolution as well. Many embedded systems consist in powerful processors, running Linux or Android. Vendors of critical real-time systems, such as in avionics, have to adapt soon: they have a hard time obtaining the predictable processors – hence extremely simple and oldish – they need, while the rest of the market evolves at a fast pace.

Need for more Adaptability More than ever, software will need to adapt to their environment. In most cases, this environment will remain unknown until runtime. This is already the case when one deploys an application to a cloud, or an *App* to mobile devices. The dilemma is the following: for maximum portability, developers should target the most general device; but for performance they would like to exploit the most recent and advanced hardware features. JIT compilers can handle the situation to some extent, but binary deployment requires dynamic binary rewriting. Our work has shown how SIMD instructions can be upgraded from SSE to AVX. Many more opportunities will appear with diverse and heterogeneous processors, featuring various kinds of accelerators.

Modern processors aggressively modify their characteristics to deliver maximum performance, while staying within the thermal envelope. The side-effect to this constant change in clock frequency (e.g. Intel Turbo Boost) is an additional difficulty in predicting what code will be best in the long run.

On shared hardware, the environment is also defined by other applications competing for the same computational resources. It will become increasingly important to adapt to changing conditions, such as the contention of the cache memories, or available bandwidth.

Faults will soon become the new *de facto* standard. The evolutions of the semiconductor industry predict an exponential growth of the number of permanent faults. Dealing the these defects will impact the performance of various components of the processor (branch predictor, caches). Furthermore, defects may appear with time, due to wear-out of the system. Handling faults at a reasonable cost will require innovative solutions.

Fortunately, optimizing at runtime is also an opportunity, because this is the first time the whole program is visible: executable and libraries (including library versions). Optimizers may also rely on dynamic information, such as actual input data, parameter values, etc.

We started addressing some of these challenges in ongoing projects such as Nano2017 PSAIC Collaborative research program with STMicroelectronics, as well as within the Inria Project Lab MULTICORE. The starting H2020 FET HPC project ANTAREX will also address these challenges from the energy perspective.

Case of Real-time systems The distinguishing characteristics of real-time system is that, beyond producing correct results, they must do so within specified deadline. Each task must be assigned an upper bound of its execution time. For critical systems, this must be safe, i.e. provably larger than any possible execution time. To avoid over-provisioning, the bound shall also be as tight as possible.

Adding a JIT compiler in a system with such a need for determinism seems unrealistic. Yet, as any other field, systems have started to evolve, from assembly language, to C, modern operating systems, and dynamic languages. We have made a first step, showing that the behavior of the code cache can be modeled. A lot remains to be done.

When hardware faults are taken into consideration (and the current lithographic process guarantees the presence of faults), tightness becomes increasingly difficult

to maintain, to the point that the estimated bound is orders of magnitude larger than the actual execution time [HP15]. Such a bound is useless. In this context, WCET estimates can be assigned a probability – a standard practice in the field of fault-tolerant systems. There is a huge opportunity for compilation, helping designers minimize the impact of faults, by applying innovative code transformations. Dynamic solutions also have a great future in this context.

Security The ever growing dependence of our economy on computing systems also implies that security has become of utmost importance. Computer systems are under constant attack, targeted by a wide range of stakeholders: from young hackers trying to show their skills, to "professional" criminals stealing credit card information, and even government agencies with virtually unlimited resources. A vast amount of techniques have been proposed in the literature to circumvent attacks. Many of them cause significant slowdowns due to additional checks and countermeasures.

Static and dynamic compilation techniques, as well as dynamic binary rewriting, will have a role to play in future security systems. The contribution will be twofold: 1) leverage and adapt the existing technology to strengthen security (as shown in our preliminary obfuscating JIT, cf. Section 3.3); 2) develop new optimization techniques to limit the overhead of security, possibly offering trade-offs.

As examples, an ANR proposal is currently under submission to study how secure codes can thwart Cyber-physical Attacks (with Télécom Paris Tech, Paris 8, Université Catholique de Louvain, and Sabancı University). We also established contacts within Inria to study the porting of an intrusion detector to a just-in-time compilation environment.

Complexity is Forever Complexity has appeared in all layers of computing systems: from hardware, through operating system, to runtimes, compilers, and applications. Assessing the performance of an application has become extremely difficult, and predicting performance has become impossible.

The relevant metric may well change during the execution of an application, from pure performance, to battery lifetime, quality of service, or power efficiency in a cloud.

Future systems will necessarily integrate feedback loops, constantly monitoring various metrics, and sending back the relevant data to a global orchestrator. It will be challenging to define to whom the feedback is sent: is it the programmer, the compiler, or the operating system?

Regardless of the actual scenario, we are firmly convinced that portability of performance will only be guaranteed when several actors cooperate: the static compiler, a JIT compiler, a runtime/operating system, and a global orchestrator. The challenge will be to define what kind of information is conveyed, between what actors, and how they should react.

Bibliography

[ABC+06] K. Asanović, R. Bodik, B. Catanzaro, J. Gebis, P. Husbands, K. Keutzer, D. Patterson, W. Plishker, J. Shalf, S. Williams, and K. Yelik. The Landscape of Parallel Computing Research: A View from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California at Berkeley, December 2006.

[Amd67] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *April 18-20, Spring Joint Computer Conference*, AFIPS '67 (Spring), pages 483–485. ACM, 1967.

[App98] Andrew W. Appel. *Modern Compiler Implementation in C*. Cambridge University Press, 1998.

[ASU88] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1988.

[Ayc03] John Aycock. A brief history of just-in-time. *ACM Comput. Surv.*, 35(2):97–113, June 2003.

[BDGR07] Florent Bouchez, Alain Darte, Christophe Guillon, and Fabrice Rastello. Register allocation: What does the NP-completeness proof of Chaitin et al. really prove? or revisiting register allocation: Why and how. In George Almási, Călin Caşcaval, and Peng Wu, editors, *LCPC*, volume 4382 of *LNCS*, pages 283–298. 2007.

[BDK+05] Shekhar Y Borkar, Pradeep Dubey, Kevin C Kahn, David J Kuck, Hans Mulder, Stephen S Pawlowski, and Justin R Rattner. Platform 2015: Intel processor and platform evolution for the next decade. *Technology@ Intel Magazine*, 3(3), 2005.

[Bel73] James R Bell. Threaded code. *CACM*, 16(6), 1973.

[BGS94] David F. Bacon, Susan L. Graham, and Oliver J. Sharp. Compiler transformations for high-performance computing. *ACM Comput. Surv.*, 26(4):345–420, December 1994.

[Bou10] Adnan Bouakaz. Predictability of just-in-time compilation. Master's thesis, University of Rennes 1, 2010.

[Bru04] Derek Bruening. *Efficient, Transparent, and Comprehensive Runtime Code Manipulation*. PhD thesis, MIT, September 2004.

[Bus11] Business Software Alliance. Eighth annual BSA global software piracy study, May 2011.

[BVZB05] Marc Berndl, Benjamin Vitale, Mathew Zaleski, and Angela Demke Brown. Context threading: A flexible and efficient dispatch technique for virtual machine interpreters. In *CGO*, 2005.

[BZA12] Derek Bruening, Qin Zhao, and Saman Amarasinghe. Transparent dynamic instrumentation. In *Conference on Virtual Execution Environments (VEE)*, pages 133–144. ACM, 2012.

[CAC+81] Gregory J Chaitin, Marc A Auslander, Ashok K Chandra, John Cocke, Martin E Hopkins, and Peter W Markstein. Register allocation via coloring. *Computer languages*, 6(1):47–57, 1981.

[Cas94] Brian Case. Philips Hopes to Displace DSPs with VLIW. *Microprocessor Report*, pages 12–15, December 1994.

[CFM+97] T. Cramer, R. Friedman, T. Miller, D. Seberger, R. Wilson, and M. Wolczko. Compiling Java just in time. *Micro, IEEE*, 17(3):36–43, 1997.

[CHH+98] Anton Chernoff, Mark Herdeg, Ray Hookway, Chris Reeve, Norman Rubin, Tony Tye, S Bharadwaj Yadavalli, and John Yates. FX!32: A profile-directed binary translator. *IEEE Micro*, (2):56–64, 1998.

[CM99] François Charot and Vincent Messé. A flexible code generation framework for the design of application specific programmable processors. In *International Workshop on Hardware/Software Codesign (CODES)*, pages 27–31, 1999.

[CN09] Christian Collberg and Jasvir Nagra. *Surreptitious software: Obfuscation, Watermarking, and Tamperproofing for Software Protection*. Addison-Wesley Professional, 2009.

[Com07] Computing Systems Consultation Meeting. *Research Challenges for Computing Systems – ICT Workprogramme 2009–2010*. European Commission – Information Society and Media, Braga, Portugal, November 2007.

[CP01] Antoine Colin and Isabelle Puaut. A modular and retargetable framework for tree-based WCET analysis. In *Euromicro Conference on Real-Time Systems*, pages 37–44, 2001.

[CPM+14] Mamadou Ciss, Nicolas Parisey, Fabrice Moreau, Charles-Antoine Dedryver, and Jean-Sébastien Pierre. A spatiotemporal model for predicting grain aphid population dynamics and optimizing insecticide sprays at the scale of continental France. *Environmental Science and Pollution Research*, 21(7):4819–4827, 2014.

[DBLM+07] Koen De Bosschere, Wayne Luk, Xavier Martorell, Nacho Navarro, Mike O'Boyle, Dionisios Pnevmatikatos, Alex Ramirez, Pascal Sainrat, André Seznec, Per Stenström, and Olivier Temam. *High-Performance Embedded Architecture and Compilation Roadmap*, volume 4050/2007 of *LNCS*, pages 5–29. 2007.

[DCRC10] Boubacar Diouf, Albert Cohen, Fabrice Rastello, and John Cavazos. Split register allocation: Linear complexity without the performance penalty. In *HiPEAC*, pages 66–80. 2010.

[DG87] J. W. Davidson and J. V. Gresh. Cint: a RISC interpreter for the C programming language. In *Symposium on Interpreters and interpretive techniques*, 1987.

[DGB+03] James C Dehnert, Brian K Grant, John P Banning, Richard Johnson, Thomas Kistler, Alexander Klaiber, and Jim Mattson. The Transmeta code morphing software: using speculation, recovery, and adaptive retranslation to address real-life challenges. In *International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*, pages 15–24, 2003.

[DYDS+10] Marc Duranton, Sami Yehia, Bjorn De Sutter, Koen De Bosschere, Albert Cohen, Babak Falsafi, Georgi Gaydadjiev, Manolis Katevenis, Jonas Maebe, Harm Munk, Nacho Navarro, Alex Ramirez, Olivier Temam, and Mateo Valero. The HiPEAC vision. *HiPEAC Roadmap*, 2010.

[EG03] M. Anton Ertl and David Gregg. Optimizing indirect branch prediction accuracy in virtual machine interpreters. In *PLDI*, 2003.

[EG04] M. Anton Ertl and David Gregg. Combining stack caching with dynamic superinstructions. In *Workshop on Interpreters, Virtual Machines and Emulators (IVME)*, pages 7–14. ACM, 2004.

[Ert95] M. Anton Ertl. Stack caching for interpreters. In *PLDI*, 1995.

[FBF+00] Paolo Faraboschi, Geoffrey Brown, Joseph A. Fisher, Giuseppe Desoli, and Fred Homewood. Lx: a technology platform for customizable VLIW embedded processing. In *ISCA*, pages 203–213. ACM, 2000.

[FCOT07] Grigori Fursin, John Cavazos, Michael O'Boyle, and Olivier Temam. MiDataSets: Creating the conditions for a more realistic evaluation of iterative optimization. In *HiPEAC*, volume 4367 of *LNCS*, pages 245–260. 2007.

[FD09] Nicholas Fitzroy-Dale. Benefits of compiler optimisation. Technical report, NICTA and the University of New South Wales, Australia, November 2009.

[FQ03] Stephen J Fink and Feng Qian. Design, implementation and evaluation of adaptive recompilation with on-stack replacement. In *International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*, 2003.

[GBEL10] Jan Gustafsson, Adam Betts, Andreas Ermedahl, and Björn Lisper. The Mälardalen WCET benchmarks: Past, present and future. *WCET*, 15:136–146, 2010.

[GMVK13] Christophe Guillon, Antoine Moynault, Cédric Vincent, and Hervé Knochel. Pseudo accurate emulation for an auto tuning optimization system. HiPEAC Computing Systems Week, Dynamic Compilation Track, May 2013.

[Guð13] Gylfi Đór Guðmundsson. *Parallelism and Distribution for Very Large Scale Content Based Image Retrieval*. PhD thesis, Université de Rennes 1, 2013.

[Hal10] Kathleen Hall. Gartner: SaaS sales will grow 16.2 % to $10.7bn in 2011. *ComputerWeekly.com*, December 2010.

[Hen06] John L. Henning. SPEC CPU2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, 34(4):1–17, 2006.

[HG83] John L Hennessy and Thomas Gross. Postpass code optimization of pipeline constraints. *TOPLAS*, 5(3):422–448, 1983.

[HLS00] Niklas Holsti, Thomas Långbacka, and Sami Saarinen. Using a worst-case execution time tool for real-time verification of the DEBIE software. In *Data Systems in Aerospace (DASIA)*, 2000.

[HM08] M.D. Hill and M.R. Marty. Amdahl's law in the multicore era. *Computer*, 41(7):33–38, July 2008.

[Hor97] Susan Horwitz. Precise flow-insensitive may-alias analysis is NP-hard. *ACM Trans. Program. Lang. Syst.*, 19(1):1–6, January 1997.

[HP15] Damien Hardy and Isabelle Puaut. Static probabilistic worst case execution time estimation for architectures with faulty instruction caches. *Real-Time Systems*, 51(2):128–152, 2015.

[HRTV11] Robert Hundt, Easwaran Raman, Martin Thuresson, and Neil Vachharajani. MAO – an extensible micro-architectural optimizer. In *CGO*, April 2011.

[IEE08] IEEE Task P754. *IEEE 754-2008, Standard for Floating-Point Arithmetic*. August 2008.

[Inr12] Inria. Towards Inria 2020 – Strategic plan 2013-2017. Technical report, Inria, 2012.

[Int14] Intel. *Intel64 and IA-32 Architectures Optimization Reference Manual*, March 2014. Order Number: 248966-029.

[KCL+99] Robert Kennedy, Sun Chan, Shin-Ming Liu, Raymond Lo, Peng Tu, and Fred Chow. Partial redundancy elimination in SSA form. *TOPLAS*, 21(3):627–676, 1999.

[L+05] Chi-Keung Luk et al. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI*, 2005.

[Lam88] Monica Lam. Software Pipelining : An Effective Scheduling Technique for VLIW Machines. In *PLDI*, pages 318–328. ACM, 1988.

[Les13] Benjamin Lesage. *Multicore architectures and worst-case execution time*. PhD thesis, Université Rennes 1, May 2013.

[LF02] Josep Llosa and Stefan M Freudenberger. Reduced code size modulo scheduling in the absence of hardware support. In *MICRO*, pages 99–110, 2002.

[LS00] Thierry Lafage and André Seznec. Combining light static code annotation and instruction-set emulation for flexible and efficient on-the-fly simulation. In *Euro-Par Parallel Processing*, pages 178–182, 2000.

[MGG+11] S. Maleki, Yaoqing Gao, M.J. Garzarán, T. Wong, and D.A. Padua. An evaluation of vectorizing compilers. In *PACT*, 2011.

[Moo65] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, pages 114–117, 1965.

[MSF+07] Pierre Michaud, André Seznec, Damien Fetis, Yiannakis Sazeides, and Theofanis Constantinou. A study of thread migration in temperature-constrained multicores. *TACO*, 4(2), June 2007.

[Muc97] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.

[NC08] Axel Naumann and Philippe Canal. The role of interpreters in high performance computing. In *XII Advanced Computing and Analysis Techniques in Physics Research*, page 65, 2008.

[NED+13] Dorit Nuzman, Revital Eres, Sergei Dyshel, Marcel Zalmanovici, and Jose Castanos. JIT technology with C/C++: Feedback-directed dynamic recompilation for statically compiled languages. *TACO*, 10(4):59:1–59:25, 2013.

[Nor03] Gregor Noriskin. Writing High-Performance Managed Applications: A Primer. In *MSDN Library*. Microsoft, 2003.

[NSS13] Surya Narayanan Natarajan, Bharath Swamy, and André Seznec. Modeling multi-threaded programs execution time in the many-core era. Technical Report RR-8453, INRIA, December 2013.

[PBB+10] L.-N. Pouchet, U. Bondhugula, C. Bastoul, A. Cohen, J. Ramanujam, and P. Sadayappan. Combined iterative and model-driven optimization in an automatic parallelization framework. In *International Conference for High Performance Computing, Networking, Storage and Analysis*, New Orleans, LA, USA, November 2010.

[Pol99] Fred Pollack. New microarchitecture challenges in the coming generations of CMOS process technologies. MICRO'32 Keynote, available at `http://research.ac.upc.edu/HPCseminar/SEM9900/Pollack1.pdf`, 1999.

[PR98] Ian Piumarta and Fabio Riccardi. Optimizing direct threaded code by selective inlining. *SIGPLAN Not.*, 33(5), 1998.

[Pro98] Todd Proebsting. Proebsting's law. `http://research.microsoft.com/en-us/um/people/toddpro/papers/law.htm`, 1998.

[PS97] P. Puschner and A. V. Schedl. Computing maximum task execution times – a graph based approach. In *RTSS*, volume 13, pages 67–91, 1997.

[Pug00] Bill Pugh. Is code optimization (research) relevant? Dagstuhl Seminar 381 – `http://www.cs.umd.edu/~pugh/IsCodeOptimizationRelevant.pdf`, September 2000.

[PVC01] Michael Paleczny, Christopher Vick, and Cliff Click. The Java HotSpot™ Server Compiler. In *Proc. of the Java Virtual Machine Research and Technology Symposium*, Monterey, CA, USA, April 2001.

[RGBW07] Jan Reineke, Daniel Grund, Christoph Berg, and Reinhard Wilhelm. Timing predictability of cache replacement policies. *Real-Time Systems*, 37(2):99–122, 2007.

[SCEG08] Yunhe Shi, Kevin Casey, M. Anton Ertl, and David Gregg. Virtual machine showdown: Stack versus registers. *ACM TACO*, 4(4), 2008.

[Sco01] Kevin Scott. On Proebsting's law. Technical Report CS-2001-12, University of Virginia, March 2001.

[SH02] Michael D. Smith and Glenn Holloway. An introduction to Machine SUIF and its portable libraries for analysis and optimization. *Division of Engineering and Applied Sciences, Harvard University*, 2002.

[SM06] André Seznec and Pierre Michaud. A case for (partially) TAgged GEometric history length branch prediction. *JILP*, 8, 2006.

[SS03] André Seznec and Nicolas Sendrier. HAVEGE: A user-level software heuristic for generating empirically strong random numbers. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 13(4):334–346, 2003.

[TFW00] Henrik Theiling, Christian Ferdinand, and Reinhard Wilhelm. Fast and precise WCET prediction by separated cache and path analyses. *Real-Time Systems*, 18(2-3):157–179, 2000.

[Val14] Cédric Valensi. *A generic approach to the definition of low-level components for multi-architecture binary analysis*. PhD thesis, Université de Versailles-St Quentin en Yvelines, 2014. 2014VERS0009.

[W+04] E. Wieprecht et al. The HERSCHEL/PACS Common Software System as Data Reduction System. *ADASS XIII*, 2004.

[WFW+94] Robert P Wilson, Robert S French, Christopher S Wilson, Saman P Amarasinghe, Jennifer M Anderson, Steve WK Tjiang, Shih-Wei Liao, Chau-Wen Tseng, Mary W Hall, Monica S Lam, and John L Hennessy. SUIF: An infrastructure for research on parallelizing and optimizing compilers. *ACM Sigplan Notices*, 29(12):31–37, 1994.

Publications

[ABB+97] Bas Aarts, Michel Barreteau, François Bodin, Peter Brinkhaus, Zbigniew Chamski, Henri-Pierre Charles, Christine Eisenbeis, John R. Gurd, Jan Hoggerbrugge, Ping Hu, William Jalby, Peter M. W. Knijnenburg, Michael F. P. O'Boyle, Erven Rohou, Rizos Sakellariou, Henk Schepers, André Seznec, Elena Stöhr, Marco Verhoeven, and Harry A. G. Wijshoff. Oceans: Optimizing compilers for embedded applications. In *Third International Euro-Par Conference on Parallel Processing (Euro-Par)*, pages 1351–1356, August 1997. LNCS 1300.

[BBB+98] Michel Barreteau, François Bodin, Peter Brinkhaus, Zbigniew Chamski, Henri-Pierre Charles, Christine Eisenbeis, John R. Gurd, Jan Hoggerbrugge, Ping Hu, William Jalby, Peter M. W. Knijnenburg, Michael F. P. O'Boyle, Erven Rohou, Rizos Sakellariou, André Seznec, Elena Stöhr, Menno Treffers, and Harry A. G. Wijshoff. Oceans: Optimising compilers for embedded applications. In *4th International Euro-Par Conference on Parallel Processing (Euro-Par)*, pages 1123–1130, September 1998. LNCS 1470.

[BBC+99] Michel Barreteau, François Bodin, Zbigniew Chamski, Henri-Pierre Charles, Christine Eisenbeis, John R. Gurd, Jan Hoogerbrugge, Ping Hu, William Jalby, Toru Kisuki, Peter M. W. Knijnenburg, Paul van der Mark, Andy Nisbet, Michael F. P. O'Boyle, Erven Rohou, André Seznec, Elena Stöhr, Menno Treffers, and Harry A. G. Wijshoff. Oceans: Optimising compilers for embedded applications. In *5th International Euro-Par Conference on Parallel Processing (Euro-Par)*, pages 1171–1175, September 1999. LNCS 1685.

[BCE+97] François Bodin, Zbigniew Chamski, Christine Eisenbeis, Erven Rohou, and André Seznec. GCDS: A compiler strategy for trading code size against performance in embedded applications. Research Report RR-3346, IRISA, 1997.

[BCE+98] François Bodin, Zbigniew Chamski, Christine Eisenbeis, Sylvain Lelait, Erven Rohou, Antoine Sawaya, André Seznec, and Jian Wang. Towards a retargetable framework for software pipelining. In *7th International Workshop on Compilers for Parallel Computers (CPC)*, pages 90–99, June 1998.

[BCRS97] François Bodin, Zbigniew Chamski, Erven Rohou, and André Seznec. Functional specification of SALTO: A retargetable system for asssembly language transformation and optimization. Technical report, LTR ESPRIT Project OCEANS, January 1997.

[BKK+98] François Bodin, Toru Kisuki, Peter M. W. Knijnenburg, Mike F. P. O'Boyle, and Erven Rohou. Iterative compilation in a non-linear optimisation space. In *Workshop on Profile and Feedback-Directed Compilation (FDO-1), in conjunction with PACT '98*, October 1998.

[BMCR99] François Bodin, Yann Mével, Stéphane Chauveau, and Erven Rohou. Porting an ocean code to MPI using TSF. In *LCPC*, pages 447–450, August 1999. LNCS 1863.

[BPR11] Adnan Bouakaz, Isabelle Puaut, and Erven Rohou. Predictable binary code cache: A first step towards reconciling predictability and just-in-time compilation. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 223–232, April 2011.

[CCFP+08] Marco Cornero, Roberto Costa, Ricardo Fernández Pascual, Andrea C. Ornstein, and Erven Rohou. An experimental environment validating the suitability of CLI as an effective deployment format for embedded systems. In *HiPEAC*, pages 130–144, January 2008. LNCS 4917.

[CER99] Zbigniew Chamski, Christine Eisenbeis, and Erven Rohou. Flexible issue slot assignment for VLIW architectures. In *International Workshop on Compiler and Architecture Support for Embedded Systems (CASES)*, Washington, D.C., October 1999.

[CGRCR04] Roberto Costa, Marco Garatti, Erven Rohou, and Stefano Crespi Reghizzi. Hardware parameters of VLIW cores and code quality factors affecting alias analysis impact. *ST Journal of Research*, 1(2):97–108, September 2004.

[COR07] Roberto Costa, Andrea C. Ornstein, and Erven Rohou. CLI back-end in GCC. In *GCC Developers' Summit*, pages 111–116, July 2007.

[CR05] Roberto Costa and Erven Rohou. Comparing the size of .NET applications with native code. In *International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 99–104. ACM, September 2005.

[CR10] Albert Cohen and Erven Rohou. Processor virtualization and split compilation for heterogeneous multicore embedded systems. In *DAC*, pages 102–107, June 2010.

[DRS96] Nathalie Drach, Erven Rohou, and André Seznec. Influence de la structure du pipeline sur les instructions de chargement. *Technique et Science Informatiques (TSI)*, 15(1):91–111, 1996.

[ECR99] Christine Eisenbeis, Zbigniew Chamski, and Erven Rohou. Flexible issue slot assignment for VLIW architectures. In *SCOPES*, September 1999.

[EEMR+14] Sara Elshobaky, Ahmed El-Mahdy, Erven Rohou, Layla AA El-Sayed, and Mohamed Nazih ElDerini. A lightweight incremental analysis and profiling framework for embedded devices. In *SCOPES*, pages 60–68. ACM, 2014.

[GCCRR02] Marco Garatti, Roberto Costa, Stefano Crespi Reghizzi, and Erven Rohou. The impact of alias analysis on VLIW scheduling. In *International Symposium on High Performance Computing (ISHPC)*, pages 93–105, May 2002. LNCS 2327.

[HEMR15] Muhammad Hataba, Ahmed El-Mahdy, and Erven Rohou. OJIT: A novel obfuscation approach using standard just-in-time compiler transformations. In *International Workshop on Dynamic Compilation Everywhere*, January 2015.

[HEMSR13] Muhammad Hataba, Ahmed El-Mahdy, Amin Shoukry, and Erven Rohou. OJIT: A Novel Secure Remote Execution Technology By Obfuscated Just-In-Time Compilation. In *Third European LLVM Conference*, April 2013.

[HRCK15] Nabil Hallou, Erven Rohou, Philippe Clauss, and Alain Ketterlin. Dynamic revectorization of binary code. In *SAMOS*, July 2015.

[LCF+07] Piotr Leśnicki, Albert Cohen, Grigori Fursin, Marco Cornero, Andrea Ornstein, and Erven Rohou. Split compilation: an application to just-in-time vectorization. In *International Workshop on GCC for Research in Embedded and Parallel Systems (GREPS), in conjunction with PACT'07*, Braşov, Romania, September 2007.

[LPR14] Hanbing Li, Isabelle Puaut, and Erven Rohou. Traceability of flow information: Reconciling compiler optimizations and WCET estimation. In *RTNS*, 2014.

[LPR15] Hanbing Li, Isabelle Puaut, and Erven Rohou. Tracing flow information for tighter WCET estimation: Application to vectorization. In *International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, 2015.

[LSRB98] Thierry Lafage, André Seznec, Erven Rohou, and François Bodin. Code cloning tracing: A new approach to trace collection. Research Report RR-3377, INRIA, 1998.

[LSRB99] Thierry Lafage, André Seznec, Erven Rohou, and François Bodin. Code cloning tracing: A "pay per trace" approach. In *5th International Euro-Par Conference on Parallel Processing (Euro-Par)*, pages 1265–1268, September 1999. LNCS 1685.

[MAB+11] Harm Munk, Eduard Ayguadé, Cédric Bastoul, Paul Carpenter, Zbigniew Chamski, Albert Cohen, Marco Cornero, Philippe Dumont, Marc Duranton, Mohammed Fellahi, Roger Ferrer, Razya Ladelsky, Menno Lindwer, Xavier Martorell, Cupertino Miranda, Dorit Nuzman, Andrea Ornstein, Antoniu Pop, Sebastian Pop, Louis-Noël Pouchet, Alex Ramírez, David Ródenas, Erven Rohou, Ira Rosen, Uzi Shvadron, Konrad Trifunović, and Ayal Zaks. ACOTES project: Advanced compiler technologies for embedded streaming. *IJPP*, 39(3):397–450, June 2011.

[NDR+11] Dorit Nuzman, Sergei Dyshel, Erven Rohou, Ira Rosen, Kevin Williams, David Yuste, Albert Cohen, and Ayal Zaks. Vapor SIMD: Auto-vectorize once, run everywhere. In *CGO*, pages 151–160, April 2011.

[OEMR14] Rasha Omar, Ahmed El-Mahdy, and Erven Rohou. Arbitrary control-flow embedding into multiple threads for obfuscation: A preliminary complexity and performance analysis. In *International Workshop on Security in Cloud Computing*, pages 51–58. ACM, 2014.

[RBCS98] Erven Rohou, François Bodin, Zbigniew Chamski, and André Seznec. Salto: un système pour la manipulation de code assembleur. In *Journées Adéquation Algorithme Architecture en Traitement du Signal et Images*, pages 113–120, January 1998.

[RBES98] Erven Rohou, François Bodin, Christine Eisenbeis, and André Seznec. GCDS: Global constraints-driven strategy. In *3rd International Workshop for Code Generation for Embedded Processors (CGEP)*, March 1998.

[RBES00] Erven Rohou, François Bodin, Christine Eisenbeis, and André Seznec. Handling global constraints in compiler strategy. *International Journal of Parallel Programming*, 28(4):325–345, August 2000.

[RBS+96] Erven Rohou, François Bodin, André Seznec, Gwendal Le Fol, François Charot, and Frédéric Raimbault. Salto: System for assembly language transformation and optimization. In *6th Workshop on Compilers for Parallel Computers (CPC)*, pages 261–272. Forschungszentrum Jülich GmbH, December 1996.

[RDN+11] Erven Rohou, Sergei Dyshel, Dorit Nuzman, Ira Rosen, Kevin Williams, Albert Cohen, and Ayal Zaks. Speculatively vectorized bytecode. In *HiPEAC*, pages 35–44, Heraklion, Greece, January 2011.

[RL10] Erven Rohou and Thierry Lafage. The pitfalls of benchmarking with applications. In *Sixth Annual Workshop on Modeling, Benchmarking and Simulation*, June 2010.

[RNSS15] Erven Rohou, Bharath Narasimha Swamy, and André Seznec. Branch prediction and the performance of interpreters – don't trust folklore. In *CGO*, February 2015.

[ROC08] Erven Rohou, Andrea Ornstein, and Marco Cornero. Compiling C to CLI for heterogeneous multicore system-on-chips. In *5th HiPEAC Industrial Workshop*, June 2008.

[ROC10] Erven Rohou, Andrea C. Ornstein, and Marco Cornero. CLI-Based Compilation Flows for the C Language. In *SAMOS*, pages 162–169. IEEE, July 2010.

[Roh94] Erven Rohou. Étude du parallélisme d'instructions. Master's thesis, University of Rennes 1, Rennes, France, September 1994.

[Roh97] Erven Rohou. Programming "multimedia" architectures for performance. Distributed Virtual Environments, organized by the French Ministry of Industry, December 1997.

[Roh98] Erven Rohou. *Infrastructures et Stratégies de Compilation pour Parallélisme à grain fin*. PhD thesis, University of Rennes 1, Rennes, France, November 1998. (in French).

[Roh09] Erven Rohou. Combining processor virtualization and split compilation for heterogeneous multicore embedded systems. In *Emerging Uses and Paradigms for Dynamic Binary Translation*, number 08441 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2009. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany.

[Roh10] Erven Rohou. Portable and efficient auto-vectorized bytecode: a look at the interaction between static and JIT compilers. In *2nd International Workshop on GCC Research Opportunities (GROW'10)*, January 2010.

[Roh12] Erven Rohou. Tiptop: Hardware performance counters for the masses. *ICPP Workshops*, 0:404–413, 2012.

[ROÖC10] Erven Rohou, Andrea C. Ornstein, Ali Erdem Özcan, and Marco Cornero. Combining processor virtualization and component-based engineering in C for many-core heterogeneous embedded MPSoCs. In *Second Workshop on Programming Models for Emerging Architectures (PMEA)*, September 2010.

[RRC+14] Emmanuel Riou, Erven Rohou, Philippe Clauss, Nabil Hallou, and Alain Ketterlin. PADRONE: a Platform for Online Profiling, Analysis, and Optimization. In *Dynamic Compilation Everywhere*, Vienna, Austria, January 2014.

[RS99] Erven Rohou and Michael D. Smith. Dynamically managing processor temperature and power. In *Workshop on Feedback-Directed Optimization (FDO-2), in conjunction with MICRO-32*, pages 73–80, November 1999.

[RWY13] Erven Rohou, Kevin Williams, and David Yuste. Vectorization technology to improve interpreter performance. *TACO*, 9(4):26:1–26:22, January 2013.

[SOR09] Gabriele Svelto, Andrea Ornstein, and Erven Rohou. A stack-based internal representation for GCC. In *First International Workshop on GCC Research Opportunities (GROW), in conjunction with HiPEAC*, pages 37–48, January 2009.

[SSNRS15] Arjun Suresh, Bharath Swamy Narasimha, Erven Rohou, and André Seznec. Intercepting functions for memoization – a case study using transcendental functions. *TACO*, 12(2):18:18:1–18:18:23, June 2015.

[vdMRB+99] Paul van der Mark, Erven Rohou, François Bodin, Zbigniew Chamski, and Christine Eisenbeis. Using iterative compilation for managing software pipeline-unrolling tradeoffs. In *SCOPES*, September 1999.

[YEMR13] Marwa Yusuf, Ahmed El-Mahdy, and Erven Rohou. On-Stack Replacement to Improve JIT-based Obfuscation – A Preliminary Study. In *International Japan-Egypt Conference on Electronics, Communications, and Computers*, Cairo, Egypt, December 2013.